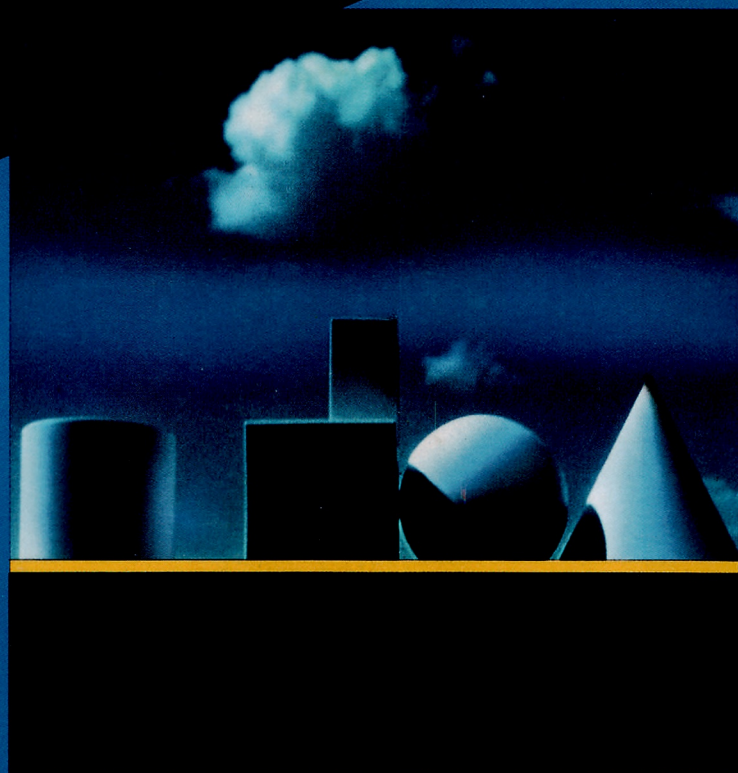
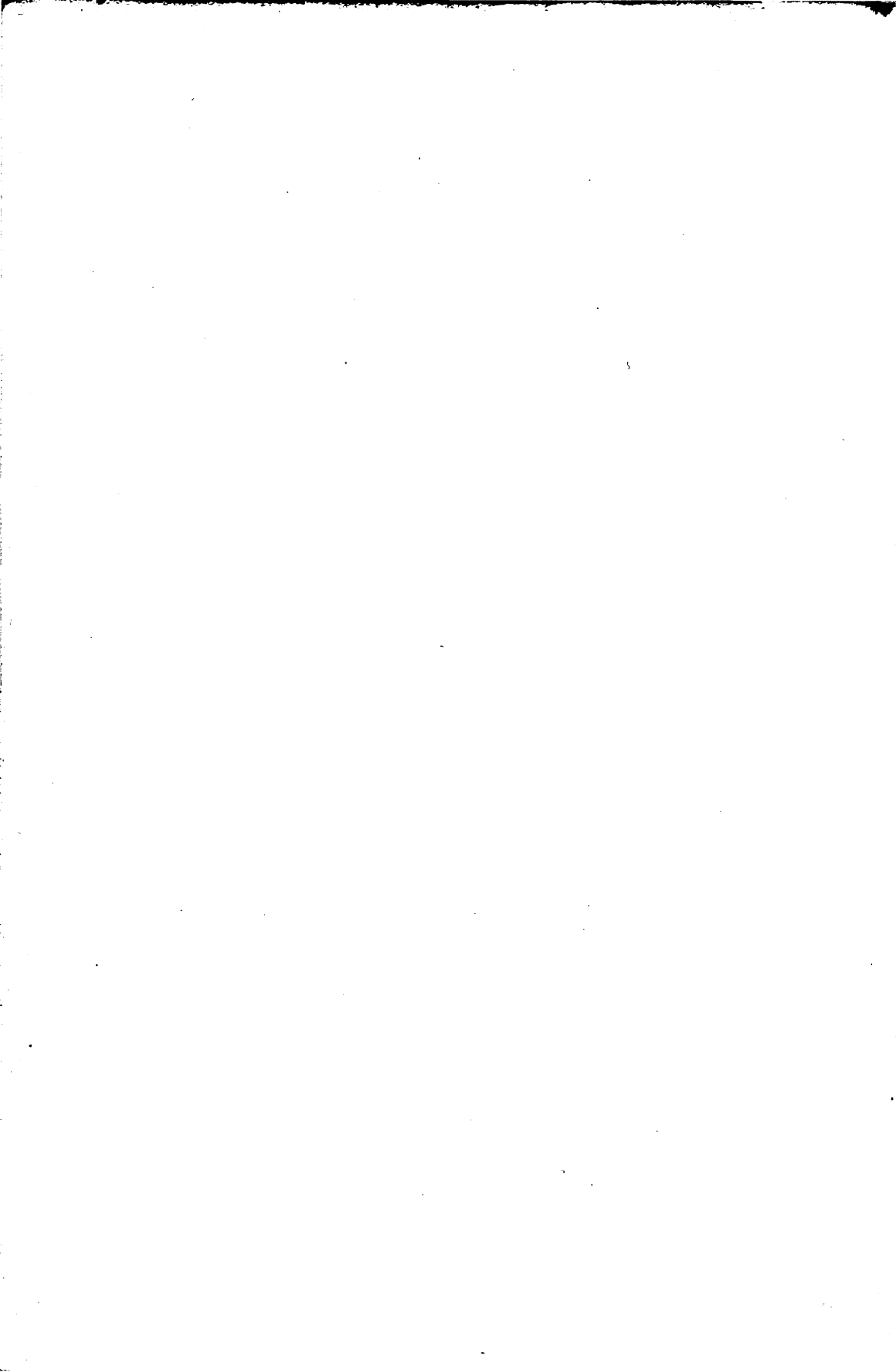


PROGRAMMING WITH
MACINTOSH^{T.M.}
TURBO PASCAL[®]



TOM SWAN



PROGRAMMING WITH MACINTOSHTM TURBO PASCAL[®]

 **Tom Swan** 

John Wiley & Sons, Inc.

New York • Chichester • Brisbane • Toronto • Singapore

*To Bill and Carmen
Never as near as friends should be*

Turbo Pascal is a registered trademark of Borland International, Inc.
IBM is a registered trademark of International Business Machines, Inc.
CP/M is a trademark of Digital Research, Inc.
Apple, Macintosh, Mac, the Apple logo, the Macintosh logo, MacWrite, MacPaint,
and MacDraw are trademarks of Apple Computer, Inc.

Publisher: Stephen Kippur
Editor: Therese A. Zak
Managing Editor: Ruth Greif
Editing, Design & Production: G&H SOHO, Ltd.

This publication is designed to provide accurate and authoritative information in regard to the subject matter covered. It is sold with the understanding that the publisher is not engaged in rendering legal, accounting, or other professional service. If legal advice or other expert assistance is required, the services of a competent professional person should be sought. FROM A DECLARATION OF PRINCIPLES JOINTLY ADOPTED BY A COMMITTEE OF THE AMERICAN BAR ASSOCIATION AND A COMMITTEE OF PUBLISHERS.

Copyright © 1987 by Tom Swan

All rights reserved. Published simultaneously in Canada.

Reproduction or translation of any part of this work beyond that permitted by section 107 or 108 of the 1976 United States Copyright Act without the permission of the copyright owner is unlawful. Requests for permission or further information should be addressed to the Permission Department, John Wiley & Sons, Inc.

Library of Congress Cataloging-in-Publication Data

Swan, Tom.

Programming with Macintosh Turbo Pascal.

1. Macintosh (Computer)—Programming. 2. PASCAL (Computer program language) 3. Turbo Pascal (Computer program) I. Title.

QA76.8.M3S95 1987

005.265

87-18980

ISBN 0-471-62417-9

Printed in the United States of America

87 88 10 9 8 7 6 5 4 3 2 1

Preface

Good tools simplify complex tasks. Good software tools help you write programs the way saber saws help carpenters build houses. You wouldn't expect carpenters to build their own saws and hammers. And you shouldn't have to spend months constructing your own software tools merely to add features that every well-designed Macintosh program must have.

You probably know that the Macintosh comes with a programmer's toolbox, which controls just about everything the computer does. The Turbo Pascal software tools in this book enhance the toolbox by adding new commands to Pascal in the form of library *units*, which you compile ahead of time and store on disk until you need them.

Chapters list the complete source code for several units. They also include detailed technical descriptions, instructions, and many programming examples. In here you'll find tools for building program shells, operating the mouse, using QuickDraw graphics, creating windows, designing dialog boxes, reading and writing disk files, transferring among applications, and clicking and dragging icons. By using these and other tools and by following the examples, you'll master the difficult art of programming the Macintosh in Turbo Pascal. You'll also find clear instructions and hints about managing memory, a controversial topic and a recognized breeding ground for program bugs.

You're ready to use these tools if you have any size Macintosh, your Turbo Pascal disks, and a fundamental knowledge of Pascal programming. You don't have to be an expert, full-time programmer to understand this material, but you should have read one or more language tutorials—or have completed a Pascal introductory course—and be able to write, compile, and run at least a small program of about 200 lines or so.

Who are you? You might be a student interested in a programming career. You might be a professional programmer with deadlines fast approaching and plenty of IBM PC or other computing experience but only a smattering of knowledge about the Mac. You might own or work for a business that needs special software but you're reluctant to gamble time and money on a custom program. You've heard

all the horror stories and you know that to do the job right, you have to do it yourself. Or you might be a serious hobbyist with more than a casual interest in programming. If you're any of these, you'll find many uses for my software tools.







I wrote the tools in this book out of necessity. When I needed a hammer, I made one. And then I used hammer to build saw, and saw to build plane, and so on, until I had a kit full of Turbo Pascal tools for building Macintosh software. It is my sincere wish that you get as much out of them as I have.

I am grateful for the contributions of many people including Teri Zak, Heather Goguen, Ruth Greif, and Claire McKean. Joe Schrader carefully critiqued the entire manuscript and made many fine suggestions. My wife and assistant, Anne Swan, helped in ways too numerous to list. I am also thankful for the professional care and expertise from everyone at John Wiley & Sons, Inc. and associates.

Tom Swan

Note to the Reader

Just before this book was printed, Apple Computer announced new system software with features to take advantage of its SE and Macintosh II computers. Although most changes have no effect on the programs in this book, a few icon symbols are different and no longer match those in Chapter 6 and elsewhere. For example, the caution, note, and stop alert symbols (see Figure 6.7 on page 260) have the new designs shown in the table here. The toolbox functions in the center of the table produce the symbols on the right for System versions 4.1 and later; they produce the symbols on the left for earlier releases.

Original icon	Toolbox function	New icon
	CautionAlert	
	NoteAlert	
	StopAlert	

Contents

one	Introducing Turbo Pascal	1
	How to Get the Most from This Book	1
	Programming by Example	2
	<i>Someone's in the Kitchen with Turbo</i>	3
	<i>About Debuggers</i>	4
	<i>Using Turbo with Single-Sided Drives</i>	6
	<i>Setting Up a Program Disk</i>	6
	<i>Starting Turbo Pascal</i>	7
	<i>Compiling to Disk</i>	9
	Turbo's Menus	11
	<i>The Apple Menu</i>	11
	<i>The File Menu</i>	12
	<i>The Edit Menu</i>	13
	<i>The Search Menu</i>	15
	<i>The Format Menu</i>	16
	<i>The Font Menu</i>	17
	<i>The Compile Menu</i>	18
	<i>The Transfer Menu</i>	20
two	Textbook Programs and Dumb Terminals	21
	Line Numbers	21
	<i>Typing and Compiling Number</i>	22
	<i>Number Play-by-Play</i>	26

Tabbing in Text	31
<i>TABS.INC Play-by-Play</i>	34
Removing Tabs from Text	35
<i>DeTab Play-by-Play</i>	38
Adding Tabs to Text	38
<i>ReTab Play-by-Play</i>	41
Converting IBM PC Programs to Macintosh	41
<i>IBM PC Identifiers Changed or Deleted</i>	42
<i>New Macintosh Identifiers Not in IBM PC</i>	
<i>Turbo Pascal</i>	51
three Turtle Graphics vs. QuickDraw	55
Turtle Graphics	57
A Star Is Born	57
<i>Star Play-by-Play</i>	59
The Twirling Turtle	60
<i>Twirl Play-by-Play</i>	62
QuickDraw Graphics	64
<i>Above the Coordinate Plane</i>	64
<i>Points and Rectangles</i>	67
A Graphics Shell	70
<i>GraphShell Play-by-Play</i>	72
<i>Saving Graphics in MacPaint Files</i>	77
Pens and Lines	77
Drawing Text	82
<i>Chars Play-by-Play</i>	86
Using Rectangles	88
Drawing Curved Shapes	90
Drawing Modes	92
Bit Maps	95
Regions	97
<i>Regions Play-by-Play</i>	99

Using screenBits	100
<i>Animate Play-by-Play</i>	104
Fractals	110
<i>Fractal Play-by-Play</i>	116
four In Any Event	121
The Parts of an Application	122
<i>Global Declarations</i>	122
<i>Program Actions</i>	123
<i>Display Handlers</i>	123
<i>Event Handlers</i>	124
<i>Initializations</i>	126
<i>Program Engine</i>	126
Developing an Application—ApShell	128
<i>ApShell Play-by-Play</i>	136
<i>ApShell Global Declarations (26–50)</i>	138
<i>ApShell Program Actions (54–175)</i>	140
<i>ApShell Display Handlers (179–216)</i>	144
<i>ApShell Event Handlers (220–339)</i>	145
<i>ApShell Initializations (343–393)</i>	153
<i>ApShell Program Engine (397–456)</i>	154
ApShell Resources	156
<i>Creating a Resource Text File</i>	156
<i>ApShell Resource Play-by-Play</i>	159
MacExtras Unit	163
<i>MacExtras Play-by-Play</i>	170
five Windows, Text, and Scroll Bars	181
Heaps Are for Keeps	181
<i>All About Handles</i>	183
<i>Molding Your Own Handles</i>	190
<i>Disposing Handles</i>	194
Multiple Windows	194
<i>MultiWind Play-by-Play</i>	200

Text in Windows	203
<i>MacStat Play-by-Play</i>	209
Picture Windows	211
<i>Picture Play-by-Play</i>	216
Text and Scroll Bars	217
<i>Reader Play-by-Play</i>	234
<i>TextUnit Play-by-Play</i>	236
 six Computer Conversations	 245
Standard File Dialogs	246
<i>SF Play-by-Play</i>	250
Dialog Item Lists	252
<i>Buttons Play-by-Play</i>	256
Dialogs in Memory	259
Alerts	259
<i>Quit Play-by-Play</i>	264
Radio Buttons	265
<i>Radio Play-by-Play</i>	271
Simple Data Entry	272
<i>Entry Play-by-Play</i>	277
Check Boxes	279
<i>Options Play-by-Play</i>	283
<i>Using Options in Programs</i>	284
Error Messages	285
<i>ErrorUnit Play-by-Play</i>	288
Testing Error Messages	290
<i>ErrTest Play-by-Play</i>	294
Data Entry Forms	295
<i>DataEntry Play-by-Play</i>	316
Dialog Tools	324
<i>DialogUnit Play-by-Play</i>	328

seven	Units as Software Tools	333
	Developing a Software Library	333
	<i>Installing Units in the Compiler</i>	334
	<i>The UnitMover Information Window</i>	335
	Transfer Tools	336
	<i>Transfer Play-by-Play</i>	338
	Let's Do Launch	340
	<i>Launcher Play-by-Play</i>	343
	Icon Tools	344
	<i>IconUnit Play-by-Play</i>	349
	Clicking and Dragging Icons	353
	<i>IconTest Play-by-Play</i>	360
	Printing Tools	361
	<i>ImageUnit Play-by-Play</i>	369
	<i>Using ImageUnit Tools</i>	374
	Putting Your Tools to Work	375
	<i>MacLister Play-by-Play</i>	392
	Bibliography	395
	<i>Software</i>	395
	<i>Books</i>	395
	Index	397

Introducing Turbo Pascal

In this book, I describe methods for writing Macintosh computer programs in Turbo Pascal. I explain tricks and techniques for using disk files, opening windows, designing dialogs, displaying icons, and driving printers. I include many software tools that you can pull out and use in your own projects. And I fully explain every statement in each listing so that you know exactly why—as well as how—the programs do what they do.

Reading the book, you'll start at the simplest level, typing in examples that run in Turbo's *textbook environment*, which does not have the familiar pull-down menus and windows, and which does not let you operate desk accessories while your programs run. Although that may seem contrary to what you've come to expect from Macintosh software, textbook programs are easy to write and often equally simple to use. They may not win any Macintosh design awards, but they come in handy when all you need is a "quick and dirty" utility. And they're ideal for short tests and simple experiments that help you to choose one programming method over another.

But my main goal in writing this book is to explain how to write fully charged Macintosh programs, ones that use all the features of the standard interface—what you see on screen in response to the things you tell the computer to do. Turbo Pascal shines by letting you write both simple textbook examples as well as these more sophisticated programs—without requiring you to change disks or to mentally switch gears. As you will learn, this is a language that lets you putter around one day but get down to serious play the next.

HOW TO GET THE MOST FROM THIS BOOK

This is a hands-on book and, to get the most from it, you'll need to get your hands a little dirty. Do type in the programs and make them run. Do modify them according to the suggestions I'll make from time to time. Do experiment. Even if all you do is type in the listings, at the very least you'll add several new programs

to your software library. And at best, you'll acquire many useful tools along with the knowledge that will help you to write your own projects. (If you would rather not type in listings, you can order them on disk by sending in the form at the back of the book.)

While that describes what this book is, you should know that it also is *not* two things. It is not a complete reference to the Macintosh toolbox, the software routines and data structures that give this computer its unique personality. And it is not a Pascal tutorial. Other books cover these subjects and I avoid duplicating their contents here. (See the Bibliography on page 395.) You don't have to purchase other books in order to use this one but you might want to pick up at least the first volume of the *Inside Macintosh* series—the Macintosh programmer's bible. Of course, you'll also need your Turbo Pascal *Users Guide and Reference Manual*, which I'll call the *Guide* from now on.

By the way, the programming in *Inside Macintosh* is almost entirely in Pascal in a dialect that, except for minor details, Turbo Pascal follows exactly. If you have any doubts about Pascal being a good choice of Macintosh programming languages, consider that Apple Computer employees featured Pascal in *Inside Macintosh*. What could be more reassuring than this direct endorsement?

PROGRAMMING BY EXAMPLE

If you haven't already read through *Inside Macintosh*, *don't* do so now. Even that well-written reference makes boring reading material—whether or not you enjoy paging through technical manuals as much as I do. Of course, *Inside Macintosh* and the *Guide* contain important information and you should read them. Rather than spending your time buried in references, though, a better plan is to read each of the following chapters, type in the examples, run the programs, and then turn to the references for more information about those subjects you don't fully understand.

By following this approach: typing examples, running programs, and reading the descriptions of how they work—and then digging into the references for more details—you'll avoid making the common mistake of trying to memorize all 1,586 pages of the four *Inside Macintosh* volumes before even writing your first program.

The truth is, it's simply unreasonable to expect to learn how to program the Macintosh by reading only technical references. You can no more accomplish your goal that way than you could learn how to cook haute cuisine by reading only nutritional guides and cookstove repair manuals. Everyone knows that one of the best ways to learn how to cook is to follow recipes and prepare dinners for guinea pig friends and family. You can follow the same approach to learn more easily how to cook up computer programs, too.

If anything, then, this is a book of recipes with ingredients for writing computer software—with special emphasis on preparing dishes in Macintosh Turbo Pascal. As the author of these recipes, I'm aware that, like your taste in food, your

taste in software might not be the same as mine. But that's unimportant. My goal is not to convince you that my programs represent ideal designs but, rather, to help you acquire the ability to season your own efforts. Of course, before learning how much salt and pepper to throw in, you should know your way around the kitchen. So put on your chef's cap and let's begin.

Someone's in the Kitchen with Turbo

If you have at least 512K memory and two double-sided disk drives or, even better, a hard disk, you'll have no trouble compiling and running the programs in this book. To help you set up your own work disks, Figure 1.1 shows the directory of my Turbo Pascal boot disk running on a Macintosh Plus with two 800K drives. On my disk, the System Folder contains the System, Finder, and ImageWriter files—a bare bones configuration that leaves the most room for storing programs. On your boot disk in your System Folder, you might have additional files that your printer, network, or desk accessories require.

Along the top row of Figure 1.1, in addition to the System Folder, are two files, RMaker and Turbo. Of course, Turbo is the Turbo Pascal editor and compiler. RMaker is a resource compiler. It reads a text file and, much in the same way the Turbo compiler reads and compiles a Pascal program, translates resources into a binary form that programs can use.

All of the fully charged examples in this book include their resources in text

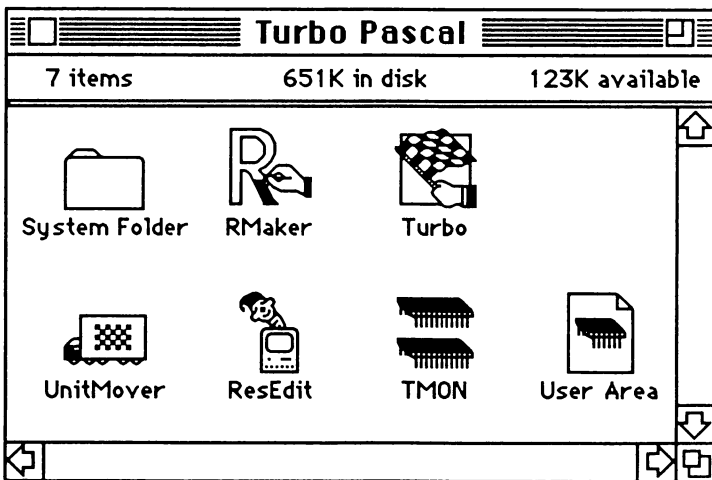


Figure 1.1 I organize my boot disk this way. To type in and run the examples in this book, you need only the three files on the top row of icons. The files on the bottom are optional though useful utilities.

form. After typing them in, you compile them with RMaker to produce a binary file which Turbo then combines with your program to produce a finished result.

The bottom row of icons in Figure 1.1 are utility programs that make programming easier—at least that's what they intend to do. These “kitchen helpers” are optional and you can remove them if you need more room on disk. UnitMover, included on the Turbo Pascal disk, operates on precompiled units, which contain common routines and data that you want various programs to share. It lets you directly install units in the Turbo compiler to customize the way it works. By doing that, you reduce the number of separate files the compiler accesses while it does its work. You can also use UnitMover to remove units from the compiler.

The three other files along the bottom row do not come with your Turbo disks. You don't need any of them to type in and run the programs in this book, but I include them here because I've found them to be useful utilities. You might want to add them to your system someday. ResEdit is Apple Computer's resource editor program. It helps you design various resources such as dialog boxes, scroll bar controls, radio buttons, windows, and other Macintosh features. TMON (The Monitor) and its associated User Area file is a debugger—a program that sits in memory looking over your program's shoulder while it runs.

About Debuggers

Debuggers like TMON let you peer into memory to look at the actual byte values that make up your program's code and data. Most debuggers have a variety of commands to trace your program's instructions in hopes of finding errors. Although many programmers swear by their debuggers, it's wise not to rely too heavily on them every time something goes wrong. There are other techniques you should try first before looking into memory and attempting to puzzle out why runaway software ran away. You'll learn many such techniques as you read this book.

One such debugger, MacsBug, is in the Misc folder on your original Turbo Pascal disk. It operates similarly to TMON but with considerably less aplomb. MacsBug has no windows—just a “dumb terminal” display that scrolls up when you type. It works well enough, though, to solve many problems and the price certainly is fair. (It comes free of charge on your Turbo disk.) To install it, just drag the MacsBug file into your System folder and reboot. You should see the message “MacsBug installed” below the familiar “Welcome to Macintosh” startup message.

Whichever debugger you decide to use, you activate it by pressing the *exception* button on the rear left side of the Macintosh case. This is the button nearest the back edge of the computer. (The Macintosh's 68000 microprocessor handles unusual conditions by a method it calls *exception processing*. In general, this process lets you interrupt a program's normal flow, go do something else, and then return to the original program—exactly what happens when you interrupt your program to use the debugger.)

Pressing the second button, the one closest to you, reboots the Macintosh—the same as turning off the power and then turning it back on. If there are no buttons on your case, you need to install them. Look in your original packing materials for a light gray plastic piece with long fingers. Your Macintosh manual tells you how to install it into the bottom vents on the rear left side.

If you don't install a debugger, pressing the exception button on a Macintosh Plus runs a ROM debugger that displays a small window and an angle bracket prompt (`>`). The program can't do very much and you should probably use TMON or MacsBug instead. If you accidentally trigger the ROM debugger, though, type `G` to return to your program. If that doesn't work, try `SM 0 A9F4` followed by `G 0`, which sets up a command to exit to the shell—usually running the Finder.

Obviously, pressing either of the two debugging switches at the wrong time can have serious consequences. Pressing the reboot button without first saving your typing throws away changes since the last time you saved your file to disk. Pressing the exception button is somewhat less dangerous. This button activates the debugger—TMON, MacsBug, or another brand. After pressing the button, you can usually return to your program without missing a beat. Type `EA` (Exit to Application) and press `RETURN` to leave MacsBug, or click TMON's exit command. Beware, though, that you might not be able to revive a badly damaged program. In that case you'd have to reboot, losing any changes you forgot to save before activating the debugger.

If you have trouble exiting from TMON when compiling and running programs directly from Turbo, select the debugger's user menu and page to the Launch command. Choose the option that launches (runs) the Finder. This works because of the way the compiler fools running programs into thinking that Turbo is actually the desktop Finder. If this method fails, follow these steps instead:

- Click open the Dump window. Type “ResumeProc” with the quotes after the message, `DUMP FROM`.
- Note the first four bytes displayed in the window.
- Click open the Regs (Registers) window. Type the four bytes from the dump window as the new PC (program counter). You do not have to type the first two bytes if they are zero.
- Click Exit to return to your program.

A different sort of debugger, HeapShow, graphically illustrates memory in a way that lets you see large amounts of the Macintosh's innards on screen—up to four megabytes at one glance! Unlike most debuggers, HeapShow installs and runs as a desk accessory. You install it in your System file with Font/DA Mover and then choose it from the Apple pull-down menu. You can even run it alongside your program to see the effects of opening dialog windows and choosing program commands.

See the Bibliography for addresses where you can write for information about these debuggers and programming utilities.

Using Turbo with Single-Sided Drives

You can use Turbo Pascal with two single-sided, 400K disk drives. In that case, your boot disk should have only the files on the top row of Figure 1.1. A good idea is to make up separate disks with UnitMover, ResEdit, and other utilities. Reboot with those disks when you need to work on resources or debug a renegade procedure.

If you have the TMON debugger, you could prepare a boot disk to load the program, and after booting, replace it with your Turbo Pascal compiler disk. You don't need the debugger disk files after loading the debugger into memory—it stays there until you reboot.

Macintosh owners with only one single-sided drive will have trouble compiling larger programs. If you have this setup, make sure your System Folder contains only the Finder, System, and ImageWriter files. Use the utility Font/DA Mover, which you received with your Turbo disk, to remove extra desk accessories and text fonts. These take up space and you don't need them to compile programs. You need only the bare bones System Folder and the Turbo file. (Font/DA Mover prevents you from removing the fonts and single desk accessory required for normal operation.) Make up other disks with RMaker, UnitMover, and other utilities. With this setup, you'll be swapping disks in and out to compile and run programs.

Another possibility for single-drive owners is to purchase a RAM disk program, which lets you simulate a second disk drive in memory. Place your Turbo file on this RAM drive and boot to a disk with nothing more than a System Folder. This will give you enough room to save and run most programs.

Setting Up a Program Disk

Many of the programs in this book expect to find certain files in named folders and volumes. You can change these requirements without doing any harm, but you'll have to watch for specific references in the listings. In all cases, such references are compiler directives (commands to the Turbo compiler) that look like the following and usually appear early in the program:

```
{ $O Programs:Shells.F: }
```

To avoid making changes to the listings, format a blank disk and give it the volume name *Programs*. When you see a name such as *Shells.F* in the listings you know by the *.F* ending that it refers to a folder. Create this folder with the Finder's

New Folder command, click its icon, type the folder name, and save your typing under the file names suggested in the chapter notes. All of this is optional. If you want to use a different setup, go ahead.

Starting Turbo Pascal

Double-click the Turbo icon to start the Turbo Pascal editor. In a moment, you'll see Turbo's menu commands in the menu bar on the top line and, in the center of the screen, an untitled, blank window with a flashing vertical bar cursor in the upper left corner (Figure 1.2).

You probably need no instructions about pulling down menus, choosing commands, typing, selecting, cutting, and pasting text. Most of you already know how to work the scroll bars and save files to disk. Neophytes who just unpacked their Macs might want to spend about an hour each with those classic programs MacPaint and MacWrite and then come back to Turbo Pascal. It won't take you long to become a Macintosh expert. Easy-to-learn is not just a sales pitch but a prime feature of this remarkable computer.

Typing and running programs in Turbo Pascal could not be simpler. But don't

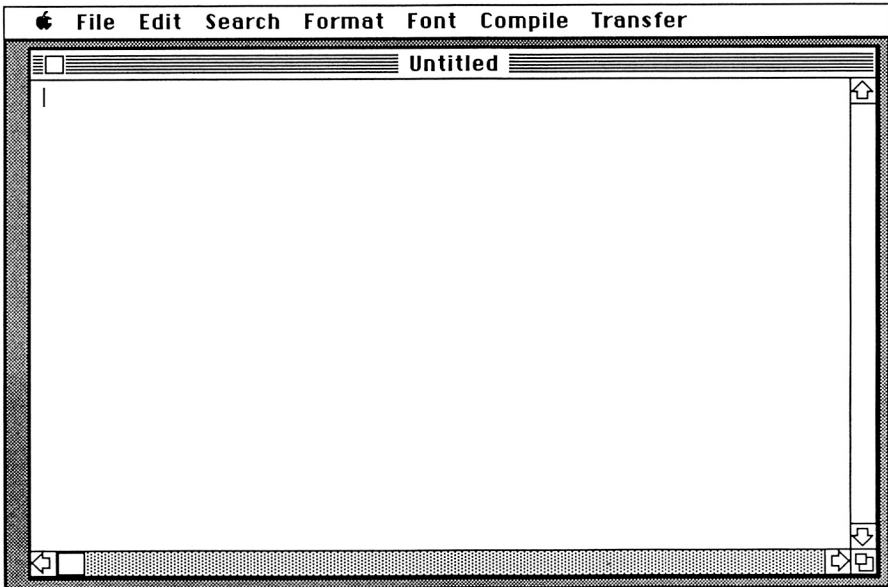


Figure 1.2 When starting a new program, the Turbo Editor gives you a blank window in which to type. At most, you can have eight windows open at one time.

Listing 1.1. NUMS.PAS

```
PROGRAM Nums;

VAR

    n : INTEGER;

BEGIN
    Writeln;
    Writeln( 'A few numbers...' );
    Writeln;
    FOR n := 1 TO 10 DO
        Writeln( n : 8 );
    END.
```

take my word for it. Prove it to yourself by typing Listing 1.1. Use the tab and backspace keys to line up columns and don't worry if your alignment isn't exactly as the listing shows.

When you're done, choose the Save As command from the File menu, and type NUMS.PAS to save your program text on disk. (The file name ending, .PAS, is merely traditional. You can name your programs anything you want but ending your Pascal source text files in .PAS tells you in a glance that this is a program listing and not something else.) After naming your text file with the Save As command, choose Save to save any changes you make. Or, choose Save As again and type a different name if you want to preserve a previously saved version. Some programmers keep all their revisions in files such as MyProg.001, MyProg.002, MyProg.003, and so on. If you do this, you can always go back to previous versions in case you find later that a change led you down a blind alley and you want to back out to the street and try another route. Of course, this also takes more disk space.

After saving your program text, choose the Run command from the Compile menu. What happened? You should have seen a brief display of numbers and then the Turbo screen again. This demonstrates a problem when writing textbook style programs such as this simple example. To make the program pause while you look at its results, insert the statements:

```
Writeln;
Write( 'Press Return...' );
ReadLn;
```

just above **END**. Then run the program again. This time, when it gets to **ReadLn**, it waits for you to press the Return key. Then it goes back to the Turbo Editor. Keep this trick in mind when typing examples from a Pascal tutorial such as my book, *Mastering Turbo Pascal*. If your programs end before you're ready, insert the above statements to make them pause before returning to the editor.

Compiling to Disk

What you just did is to *compile* a Pascal program in source code form—meaning text—to binary code, or machine language. When you do this with Turbo's Run command, your program operates in a sort of piggy-back fashion on top of Turbo Pascal, which remains in memory along with the program code and text.

Another way to compile and run programs is to choose the Compile menu's To Disk command. This compiles programs into binary code the same way the Run command does but stores the result on disk as an application—a program that you run directly from the Finder. On disk, the name of your program is the same as the name you use after the **PROGRAM** identifier, in this example, **Nums**. To change the name of your program's code disk file, insert an Output *compiler option* as the first line in your program. The option has the form:

```
{ $O fileName }
```

Replace *fileName* with the name you want the compiler to use. You can also specify volume names and folders by putting them in front of the name and separating the parts with colons. If you have the disk volume *MyDisk* and a folder *Programs*, save the **Nums** application code in that folder by writing this as the first line:

```
{ $O MyDisk:Programs:Nums }
```

If you leave out **Nums** and just end the file specification with a colon, the output goes to the volume and folder you specify, but it has the same name as the identifier after **PROGRAM**, just as it does when you don't use an Output compiler option. This is the method many examples in this book use to send their code to specific folders on disk.

When compiling to disk code files, there are two ways to run a program. These are:

- Quit Turbo and double-click the application icon.
- Choose the File menu's Transfer command and select your program (or any other application).

When you transfer to another application, Turbo does not stay in memory. After you quit your program, you return to the Finder and must restart Turbo Pascal to compile another program. Of course, your program may have its own Transfer command, in which case you could transfer back again to Turbo. (Chapter 7 explains how to add a Transfer command to your own projects.)

Figure 1.3 shows the relationship among the Finder, Turbo Pascal, and your program. The shaded lines enclose items in memory depending on the compiling method you use. The arrows show the order in which you use the items. For exam-

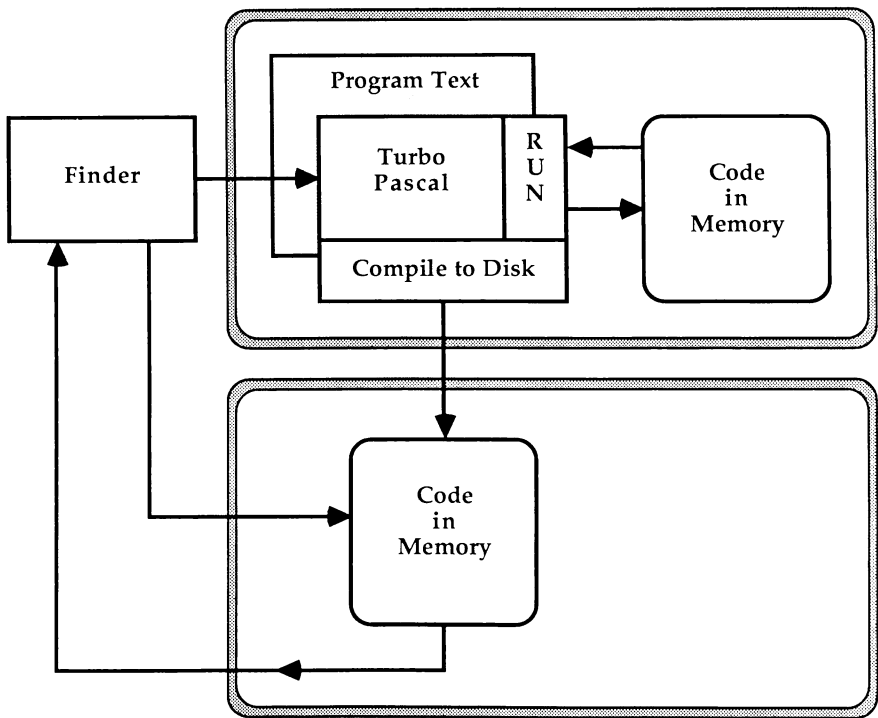


Figure 1.3 There are two ways to run Pascal programs in Turbo: directly from the editor or from the Finder. The first method takes more space because your program text, the compiler, and your code exist in memory together. The second method opens more memory to your program, which no longer has to share space with other items.

ple, from the Finder, you can run a code file—an application—or you can run Turbo, which can run code two different ways. As you can see when you compile to memory, Turbo Pascal and your program's source text and binary code are all in RAM at the same time. When you compile to disk, only your program code is in memory.

Because the editor lets you open eight windows, you can have up to that many programs in memory and switch among them simply by bringing a window to the front and choosing the Run command. But remember that this might cause you to run out of RAM—especially if you have a smaller Macintosh or if you have large debuggers and RAM disks, which occupy even more memory space. For that reason, it's probably best, at least for the examples in this book, to work on one program at a time. Turbo's in-memory compiling ability is extremely useful but you must remember that it cannot work under all possible conditions.

TURBO'S MENUS

The Turbo system has eight pull-down menus. This section gives brief descriptions of what every command does. Figures 1.4 through 1.13 illustrate each menu, adding brief descriptions to the right of the commands. Of course, the *Guide* details every Turbo feature and command. Rather than duplicate this information here, the following notes suggest several hints that you might not pick up on your own.

The Apple Menu

The only command here that belongs to Turbo Pascal is the first, About Turbo. When you choose it, you see Borland International's commercial message. (In later chapters, I'll show you how to add similar commercials to your own programs.)

The other commands in the Apple menu are desk accessories, some of which come on your Turbo disk. The Alarm Clock, Control Panel, Key Caps, and Note Pad are probably old friends. Even if you've been using your Macintosh for a brief time, these standard accessories need little introduction.

But the reason I list this menu here is to point out the HeapShow and miniDos accessories, which do not come with Turbo Pascal. I mentioned HeapShow earlier—it's the debugger that displays large amounts of memory. There are two versions, one loading into the system heap (HeapShowS) and the other into the application heap (HeapShowA). (If you don't know what the heap is, you'll learn more about it in future chapters.) Although you can use either version, HeapShowA is probably best. It displays the memory area containing your program's data structures along with its code if you run it from disk.

The miniDos accessory is a public domain program, available through most user groups. (If you can't find this program, try looking through magazines for


	
About Turbo...	- Display commercial message
<hr/>	
Alarm Clock	- Standard desk accessory
Control Panel	- Standard desk accessory
HeapShowA	- Debugger in application heap
HeapShowS	- Debugger in system heap
Key Caps	- Keyboard characters
miniDos	- Public domain disk file utility
Note Pad	- Standard desk accessory

Figure 1.4 Turbo's Apple menu.

sources that distribute public domain software.) Once you use a utility like miniDos, you won't want to give it up. It adds renaming, deleting, and other disk file commands to programs such as Turbo Pascal that don't have these abilities. If I knew the author of this program, I'd certainly give credit here. It's a valuable program.

The File Menu

The File menu's New command (Figure 1.5) opens an untitled window—use Save As to give it a name. Open reads an existing text file, ready to compile or edit. Close removes a window from the screen. If you made changes, it asks whether you want to save your file to disk. Save writes your file to disk using the name you last supplied to Open or to Save As. If you didn't specify a name, Save asks you to supply one. All of these commands operate similarly in many other Macintosh programs.

Not so familiar is the Open Selection command. To use it, select a file name by clicking and dragging the mouse over the characters in the active window. Or double-click the mouse to select an entire word. For example, if you have the compiler *Include* directives,

```
{ $I MyProg1 }
{ $I MyProg2 }
{ $I MyProg3 }
```

File		
New	⌘N	- Open new window (up to 8)
Open...	⌘O	- Open existing text file
Open Selection	⌘P	- Open file name selected in text
Close	⌘.	- Close active window
Save	⌘S	- Save changes to disk
Save As...		- Name a file before saving
Page Setup...		- Prepare printer options
Print...		- Print text in active window
Edit Transfer...		- Edit entries in the transfer menu
Save Defaults		- Save Turbo options
Transfer...	⌘T	- Transfer to another application
Quit	⌘Q	- Quit Turbo and return to Finder

Figure 1.5 Turbo's File menu.

you can double-click on MyProg1, 2, or 3, and then choose Open Selection to read that file from disk into a new window. This is handy when your program is in separate files and you want the compiler to include those pieces as though they were one. (That's what the Include directive does—it loads a separate text file as if that text were in the main file at that position.) Using the Open Selection command is easier than opening separate files one-by-one.

Page Setup and Print are standard commands to configure your printer and print text from the active window. Before printing Pascal listings, though, you might want to wait until you type in MacLister in Chapter 7. The program uses the ImageWriter printer's native text mode, which is faster than Turbo's standard printing ability.

The Edit Transfer command lets you add or subtract the names that appear in the Transfer menu (see Figure 1.13 on page 20). By installing application names in this menu, you can transfer to them by choosing their names from this menu. This is particularly useful when switching among various utility programs such as ResEdit and RMaker. You can also compile your program to disk, insert its name into the Transfer menu, and select it just as you do other commands.

Save Defaults writes to disk changes you make to the Turbo Editor. It saves the names you type with the Edit Transfer command plus the contents of the two Options dialogs from the Edit and Compile menus. After setting the options the way you want them and choosing Save Defaults, Turbo uses those same selections the next time you start it from the Finder.

The Transfer command brings up a standard file dialog, from which you select the name of another application you want to run. It does the same thing as typing an application name with Edit Transfer and then using the Transfer menu to run that program. If you transfer to the same program many times, it's probably best to install its name in the Transfer menu rather than use the Transfer command, which simply takes more steps. Quit—of course—quits Turbo and returns to the Finder.

The Edit Menu

Undo, Cut, Copy, Paste, and Clear (Figure 1.6) undoubtedly hold no mysteries for most Macintosh owners. Undo is reasonably smart and usually can reconstruct an entire line after you've made changes to it. But if you make changes to one line and then use the mouse to position the cursor elsewhere, you might lose the ability to undo what you previously could have undone.

Shift Left and Shift Right are two commands that ought to be required by law in other Pascal program editors. (They're easiest to use by typing the command and left or right square bracket keys—I rarely choose them from the menu.) To shift lines, select them by clicking and dragging the mouse. Be careful to select only entire lines—the commands do not work if you select only a partial line. (If you do it correctly, your selected text appears as a perfectly rectangular dark block with

Edit		
Undo	⌘Z	- Undo most recent editing change
Cut	⌘H	- Cut selected text
Copy	⌘C	- Copy selected text
Paste	⌘V	- Paste cut or copied text at cursor
Clear		- Erase text (can't later be pasted)
Shift Left	⌘[- Shift selected lines left
Shift Right	⌘]	- Shift selected lines right
Options...		- Change editing options

Figure 1.6 Turbo's Edit menu.

no over- or underhangs.) After selecting the text you want to move, shift the lines left or right, moving one character position for every keypress. This is most handy for Pascal listings where you frequently change indentations to show nesting levels in **WHILE** loops and in other situations.

The Edit menu Options command brings up the dialog in Figure 1.7. It lets you change the tab width and tell Turbo if you want it to automatically indent lines and whether or not to start with a blank, untitled window. (There's another Options command in the Compile menu—don't let it confuse you.) I set tab widths to 3 and you should do the same if you want your listings to look like the examples in this book. But you are free to use any setting you want—it doesn't affect the way programs run.

Switch on Auto Indent if, after pressing Return, you want Turbo to place the cursor directly below the first non-blank character in the line above. (It's normally on.) The advantage when typing Pascal listings is that you don't have to tab to the beginning of each new line in order to maintain indentation levels. The disadvan-

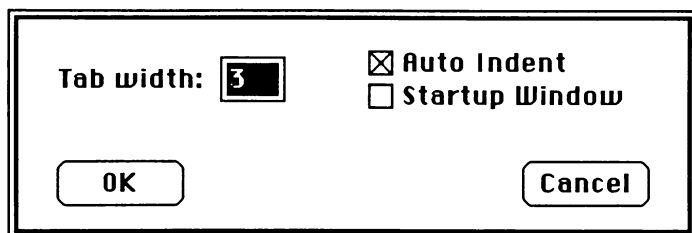


Figure 1.7 Turbo's Edit menu Options command displays this dialog window.

tage is that you have to press the backspace key or use the mouse to move the cursor to the left. Try both settings to see which you prefer. (I keep it on.)

Switch the Startup Window option off and Turbo will not automatically open a new window every time you start it. This is the setting I use. It avoids having to close the untitled window every time I start Turbo simply to edit another file. (This is a good option to consider for your own programs. How many times have you started MacWrite or MacPaint only to have to close its window in order to load a different file?)

Remember to choose the Save Defaults command from the File menu to save your options. That way, Turbo remembers your settings for the next time.

The Search Menu

The menu in Figure 1.8 may be somewhat misnamed—only the first three commands do any searching. Find lets you search for text fragments. After you choose it, a dialog window appears (not shown here). Type the string you want to find and click OK to begin searching from the current cursor position down towards the end of your text.

If you want to find only whole words—surrounded by blanks or punctuation—click that option. This is helpful when you have a lot of words such as **ScrnDump** and **ClearScrn** and you want to find **Scrn** all by itself. Click the Case Sensitive button to find only exact upper- and lowercase spellings of words. With this button off, **While**, **while**, and **WHILE** are no different to the search operation.

One trick to remember is that you can double-click any word in your text and then choose the Find command. Doing that automatically loads the selected word as the next search fragment.

Find Next is the same as Find but repeats the same search without making you type the fragment again. Get in the habit of typing Command-D to locate the next occurrence of words.

Choose Change when you want to replace one word with another. (You can

Search		
Find...	⌘F	- Find text fragment
Find Next	⌘D	- Find same text fragment
Change...	⌘R	- Find fragment(s) and change
Home Cursor		⌘H - Move cursor to top of window
Window		⌘W - Switch to another window

Figure 1.8 Turbo's Search menu.

double-click the search fragment just as you did with the Find command.) Press the Tab key to advance to the line that says Change To: and type whatever new text you want. If Turbo finds at least one of your search fragments, it asks you for permission to change it (click Yes or No), whether you want to replace all such fragments without being asked for permission from now on (click All), or if you want to stop searching now (click Cancel).

Type Command-H to home the cursor—meaning to send it to the top left corner of the active window. This also redisplay text from the first line. Before using the Find and Change commands, first remember to home the cursor. Searching always proceeds from the cursor down toward the end of the text—despite what portion of text you see in the window. If the cursor is at the end of your document and you try to Find something, you probably will hear a beep, indicating that Turbo didn't find your search fragment. This can be confusing if you are viewing your text from the top but you left the cursor at the bottom. Remember to home the cursor and you won't have that problem. (Sometimes you may want to search from, say, the middle of your text and ignore words above the cursor. In that case, don't home the cursor before starting a search.)

The final command in the Search menu, Window, rapidly changes from one window to another. When you have several windows open at the same time, it's easier to type Command-W than it is to drag windows aside and click inside their borders to activate the window you want. The disadvantage of the Window command is it doesn't let you choose which window you want to see next—it just cycles through them all. It works fast enough, though, that this is more of a minor annoyance than a major problem.

The Format Menu

The upper part of the Format menu (Figure 1.9) affects windows; the lower part affects text inside windows. When you have more than one window open, you can Stack them on top of each other like shingles on a roof. Or you can Tile them, placing each window in its own area on the screen. With tiling, the more windows you have, the smaller each becomes.

I prefer tiling multiple windows and selecting the one I want by clicking its title bar. Then I use the Zoom Window command to expand that window to full size. After making changes to the text in that window, I again choose Zoom Window to reduce the window back to its original size. That uncovers the other tiled windows, letting me select another one to zoom. If you open four or six windows now and Tile them, you'll see what I mean.

A shortcut is to double-click any window title bar, which does the same thing as zooming. This is much easier than choosing the Zoom command, and it works in both directions—that is, from little window to full size and back again.

The font sizes in the bottom part of the Format menu let you choose the size of text in windows. For programming, you probably should use the default setting

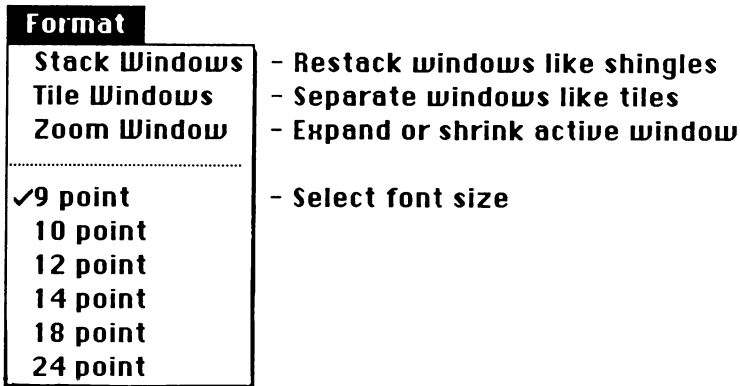


Figure 1.9 Turbo's Format menu.

of 9 points unless you have trouble seeing such small text. Another good choice is 12 points. Other sizes don't look very good in the standard Monaco text font.

If you change the point size, it affects only the current window. If you then load another file, its text uses the new size setting. But remember that changing font sizes, unlike some other editors such as MacWrite, never changes anything on disk. The next time you load a file, its text displays in whatever point size is now in effect. Remember to save the default settings if you change point sizes and want Turbo to use that size from now on.

The Font Menu

There's not much to say about the menu in Figure 1.10. You can select different fonts in which to display text although the default Monaco font is the one most programmers prefer. This font, whose name is a play on words—it's a Monospaced font—makes it easy to line up columns, a fact that makes indented languages like Pascal easy to read. The other fonts are proportionally spaced and, therefore, don't produce as good-looking listings.

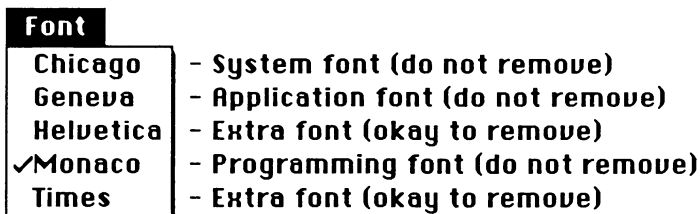


Figure 1.10 Turbo's Font menu.

The Monaco font does have a few annoying features that detract from its usefulness in programming. For example, the lowercase l and the uppercase I are identical—a disaster for programmers who tend to be fussy about single bits, not to mention whole characters. Some people use a utility like ResEdit to add a hat and foot to uppercase I's and put a slash through zeros.

If you need the disk space, use the Font/DA Mover utility to remove the Helvetica and Times fonts (in their various point sizes). You don't need them to write programs. Be careful, though, not to remove fonts that *other* programs need—including your own.

The Compile Menu

You'll probably use the Compile menu (Figure 1.11) the most often. I already explained the Run and To Disk commands. To Memory is similar to Run in that it compiles your program directly into Macintosh memory. After you do that, choosing Run runs the program without recompiling. But if you make any changes between the time you compile to memory and choose Run, Turbo recompiles your program anyway.

It may seem redundant to have the To Memory command when Run also compiles to memory. But there are two times when you need it. When you have a program in two or more pieces and use compile Include directives to compile those pieces as explained earlier, choose Run to compile and all goes well. But if you then make a change to a file that is not the main program and choose Run, Turbo knows you didn't change the main text and it just reruns your same program—it doesn't recompile it and include the changes you just made to the other pieces. The compiler doesn't know that text in other windows is related to the text you tell it to compile. For this reason, you first have to compile to memory after changing a supporting text file and then choose Run to run the program.

Compile		
Run	⌘R	- Compile and/or run program
To Memory	⌘M	- Compile only--do not run
To Disk	⌘K	- Compile to disk file
Check Syntax	⌘Y	- Check for errors in program
Find Error	⌘E	- Find fault of run-time error
Get Info	⌘I	- Display facts about program
Options...		- Change compiling options

Figure 1.11 Turbo's Compile menu.

The other time to compile to memory is when you design your own units, containing precompiled routines and data that you want other programs to share. As with Include files, if you change something in a unit after running your program, Turbo won't recompile that same program unless you make a change to it. To force Turbo to read the new programming in your modified unit, select To Memory and then Run. (The To Disk command doesn't have this same idiosyncrasy, by the way. It always compiles your program from scratch.)

The Check Syntax command compiles the program in the frontmost window, but it does not generate machine language code. Because it skips this step, it is faster than Run or To Memory. Check Syntax can't tell if your program will run correctly; it tells only whether you wrote it according to Pascal syntax, the rules that define the language and how its parts go together.

Use the Find Error command to locate where in your program an error occurred while the program was running. If you receive a "Bomb Box" error and click the Resume button, Find Error locates the instruction that caused the problem. This assumes, though, that you are able to return to the editor. Some errors won't let you back into Turbo and, in that case, you have no choice but to reboot.

Choose Get Info for some information about your program. A window tells you how many bytes your text occupies in memory, how many lines it contains, whether you compiled the program and, if you did, the size of its code and data sections. It also tells you how much memory you have available in the heap—a term that refers to an area of memory of which we'll be seeing a lot in future chapters.

The final Compile menu command, Options, brings up the dialog in Figure 1.12. The Symbol table default setting of 32K is plenty for all the examples in this book. To conserve memory, lower this value. While compiling, if you ever receive the error message "Too many symbols," set it a little higher.

The Auto Save Text option is very helpful and I suggest you turn it on. (Normally, it is off.) That way, every time you choose Run from the Compile menu, Turbo automatically saves to disk text in all windows in which you made changes. The first time you run a program but forget to save your text—and have to reboot due to a program bug—you'll appreciate the value of the Auto Save feature.

The Default Directories tell Turbo where to look for various files. Each of the letters to the left of the long boxes stands for one type of file according to this plan:

- U—Unit code files
- I—Include files
- R—Resource files
- L—Assembly language .REL files
- O—Output files

For example, if you want to load Include files from the volume PROGRAMS:LIBRARY:, then type that in the box next to the letter I. This lets you keep files of one type together in folders and is particularly useful if you have a

Symbol table K-Bytes ☒ **Auto Save Text**

Default Directories:

\$U

\$I

\$R

\$L

\$O

OK **Cancel**

Figure 1.12 Turbo's Compile menu Options command displays this dialog window.

hard disk. Rather than cluttering a main directory with all kinds of files, you can put your units in one folder, your resources in another, and send your compiled code to yet another folder.

Turbo saves these options along with the Edit menu options when you choose the File menu's Save Defaults command.

The Transfer Menu

The commands in the Transfer menu (Figure 1.13) are completely up to you. Anything you type using the Edit Transfer command in the File menu shows up in the Transfer menu. To transfer to another program, just choose its name from this menu.

Be aware that Turbo doesn't check whether the names you type are the actual names of real applications. If you type Rumpelstiltskin, Turbo presents that name in the Transfer menu. If you try to transfer to a file that doesn't exist, you receive a "File not found" error. If that happens, click the mouse to return to Turbo.

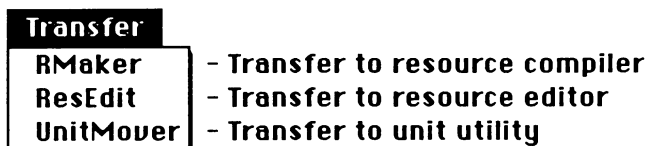


Figure 1.13 Turbo's Transfer menu.

Textbook Programs and Dumb Terminals

When you run simple Pascal programs, Turbo displays a simulated dumb terminal, similar to the computer terminals of old with no processing ability and, in some cases, unable even to position the cursor, erase lines, or do other screen operations. The Turbo dumb terminal is a bit smarter than that, but it's still no fancy CRT. Don't expect much from it.

Despite being primitive, though, Turbo's dumb terminal display is useful for writing short examples such as the ones you might find in a Pascal textbook—hence the name, *textbook program*. You can also use this display for experiments and utilities that do simple jobs when you don't care whether you can access desk accessories or use windows.

If you are typing programs from a Pascal tutorial, the last part of this chapter lists keywords that are different on the Macintosh and IBM PC versions of the compiler. This list, along with your other references, helps you to weed out and rewrite statements that work on only one or the other computer.

To demonstrate that textbook programs do have value, though, following are three programs I used frequently to help prepare this text and to convert program examples and tests created with other text editors.

LINE NUMBERS

Line numbers in program listings may seem out of place to experienced Pascal programmers, but I include them for several reasons in all examples (except for the short program in Chapter 1). They make it easy for me to refer to specific lines and, therefore, they make it easier for you to find those lines without hunting through the text, looking for words and phrases. They also prevent accidentally leaving lines on the cutting floor while preparing this book for publication. If the line numbers are in sequence, you can be sure that the listings are intact. Remember,

though, that the line numbers are not part of the program and you must exclude them while typing. Type only the text to the right of the numbers and colons in the left column.

And that brings us to the first example of a textbook style program—the same program I used to add the line numbers to the listings in this book. The program demonstrates also how to read and write disk text files.

In the following sections, I describe how to type in, compile, and run Number. I then explain each line of the program in a “play-by-play” description, pointing out details that will help you to write similar examples of your own. I follow this same format for most of the examples to come.

Typing and Compiling Number

Type in Listing 2.1, shown on page 24. (Remember not to type the line numbers.) Use the Tab key to maintain approximately the indentation in the listing. But don't worry if your typing doesn't match exactly the printed copy here—Pascal ignores indentation when it compiles a program.

Save your program as NUMBER.PAS using the File menu's Save As command. Compile to memory or to a disk code file. (If you have the Auto Save option on, you can simply Run. In that case, Turbo automatically saves your text to disk. If

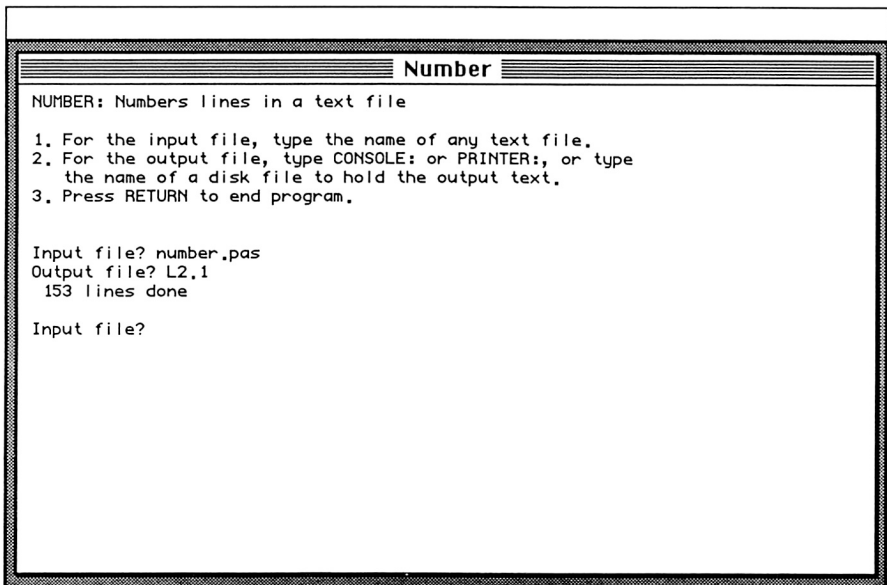


Figure 2.1 The Number program (Listing 2.1) uses Turbo's dumb terminal window, which has a blank line at top instead of the usual pull-down menus.

you haven't given your file a name, Turbo will ask for one before running the program.)

Figure 2.1 shows Number in action and illustrates Turbo's dumb terminal display. Notice that at the top there are no menus or desk accessories, just a blank space. Although the window has a drag bar along the top, you cannot move it by dragging with the mouse. It's cemented in place.

The window title is the name of the program now running. Below the title bar appears the program's output—the strings and other information you insert in **Write** and **Writeln** statements. Such text always appears in the default 9-point Monaco font—you cannot change to a different font or point size. Inside the window, you can display up to 25 rows of 80 columns, exactly the same dimensions as a standard IBM PC display as well as many other computer displays and terminals.

Number starts by identifying itself and listing a few instructions. It then requests input and output file names. If it finds the input name you type and if it has no trouble creating the output file, it reads all lines of text from the input, adds line numbers, and writes the result to the output file. If you accidentally type the name of an existing file for your output, Number displays this message:

```
ERROR: Bad or duplicate file name
```

In that case, type a different name or remove the existing file and try again. If you don't receive an error, the program runs, reporting as it goes how many lines are done. When finished, it requests another file. Type a different name or press Return to end. This takes you back to the Turbo editor or to the Finder, depending on the way you originally ran the program.

This format—typing input and output file names and pressing the Return key to end the program—is archaic and contrary to the Macintosh way of doing things. But the 153-line program, short as Macintosh programs go, does its job well enough. To do the same thing and include pull down menus and desk accessories might take four or five times that number of program lines. That's not to argue against such Macintosh features but to suggest that, when designing programs, you consider whether it is worth the trouble of always following the Macintosh interface guidelines. At times, a quick and dirty utility like Number is adequate if it does what you need it to do.

You might wonder how to number the lines in text files inside other volumes and folders. To do that, add volume and folder names with colons in between. If you have a disk named WORK and a folder named LISTINGS, you could type:

```
Output file? WORK:LISTINGS:LINES.TXT
```

Apple experts recommend never requiring people to type volume and folder names that way but, when using Turbo's dumb terminal interface, you don't have any choice. There's no easy way to use the standard file name dialog to select files by clicking their names. Of course, I'll show you how to do exactly that when we get to writing fully charged Macintosh examples.

Listing 2.1. NUMBER.PAS

```

1: PROGRAM Number;
2:
3: (*
4:
5:  * PURPOSE : Number lines in a text file
6:  * SYSTEM  : Macintosh / Turbo Pascal / Textbook interface
7:  * AUTHOR   : Tom Swan
8:
9: *)
10:
11:
12: TYPE
13:
14:   String64 = String[64];
15:
16: VAR
17:
18:   inFile, outFile : TEXT;
19:   inName, outName : String64;
20:   done : BOOLEAN;
21:
22:
23: FUNCTION Verified( message : String64 ) : BOOLEAN;
24:
25: { TRUE if Y typed, else false }
26:
27:   VAR
28:
29:     ch : CHAR;
30:
31:   BEGIN
32:     Write( message, '? (y/n) ' );
33:     Readln( ch );
34:     Verified := ( ch = 'Y' ) OR ( ch = 'y' )
35:   END; { Verified }
36:
37:
38: FUNCTION Duplicated( name : String64 ) : BOOLEAN;
39:
40: { TRUE if disk file already exists or permission granted to erase }
41:
42:   VAR
43:
44:     tempFile : TEXT;
45:
46:   BEGIN
47:     IF Pos( ':', name ) = Length( name )
48:     THEN
49:       Duplicated := FALSE { Not a disk file if it ends with : }
50:     ELSE
51:       BEGIN
52:         {$i-} Reset( tempFile, name ); {$i+}
53:         IF IoResult <> 0
54:         THEN
55:           Duplicated := FALSE { File not found }
56:         ELSE
57:           BEGIN
58:             Duplicated :=
59:               NOT verified( Concat( 'Remove old ', name ) );
60:             Close( tempFile ) { Close the temporary file }
61:           END { else }
62:         END { else }
63:       END; { Duplicated }
64:

```

```

65:
66: FUNCTION FilesOpened( VAR done : BOOLEAN ) : BOOLEAN;
67:
68: { Return true if input and output files opened. }
69: { Set done to TRUE if no input file name entered. }
70:
71:     VAR
72:
73:         okayFlag : BOOLEAN;
74:
75:     BEGIN
76:         okayFlag := FALSE;      { assume the worst case }
77:         Write( 'Input file? ' );
78:         Readln( inName );
79:         done := length( inName ) = 0;
80:         IF NOT done THEN
81:             BEGIN
82:                 { $i- } Reset( inFile, inName ); { $i+ }
83:                 IF IoResult = 0 THEN
84:                     BEGIN
85:                         Write( 'Output file? ' );
86:                         Readln( outName );
87:                         IF length( outName ) > 0 THEN
88:                             IF NOT Duplicated( outName ) THEN
89:                                 BEGIN
90:                                     { $i- } Rewrite( outFile, outName ); { $i+ }
91:                                     okayFlag := ( IoResult = 0 );
92:                                 END; { if }
93:                             IF NOT okayFlag
94:                                 THEN close( inFile )
95:                             END { if }
96:                         END; { if }
97:                         FilesOpened := okayFlag
98:                     END; { FilesOpened }
99:
100:
101: PROCEDURE NumberLines;
102:
103: { Read lines from inFile, attach line numbers, and write to outFile }
104:
105:     CONST
106:
107:         bs = #8;      { Backspace control char }
108:
109:     VAR
110:
111:         oneLine : String[255];
112:         lineNo  : INTEGER;
113:
114:     BEGIN
115:         lineNo := 0;
116:         WHILE NOT EOF( inFile ) DO
117:             BEGIN
118:                 lineNo := lineNo + 1;
119:                 Readln( inFile, oneLine );
120:                 Writeln( outFile, lineNo:3, ': ', oneLine );
121:                 Write( lineNo:4, bs, bs, bs, bs ) { display line numbers }
122:             END; { while }
123:             Writeln( lineNo:4, ' line(s) done' )
124:         END; { NumberLines }
125:
126:
127: PROCEDURE DisplayInstructions;
128:

```

(continued)

```

129: BEGIN
130:   Writeln( 'NUMBER: Numbers lines in a text file' );
131:   Writeln;
132:   Writeln( '1. For the input file, type the name of a text file.' );
133:   Writeln( '2. For the output file, type CONSOLE: or PRINTER:, or' );
134:   Writeln( '   the name of a disk file to hold the output text.' );
135:   Writeln( '3. Press RETURN to end program.' );
136:   Writeln
137: END; { DisplayInstructions }
138:
139:
140: BEGIN
141:   DisplayInstructions;
142:   REPEAT
143:     Writeln;
144:     IF FilesOpened( done ) THEN
145:       BEGIN
146:         NumberLines;
147:         Close( inFile );
148:         Close( outFile )
149:       END ELSE
150:       IF NOT done
151:         THEN Writeln( 'ERROR : bad or duplicate file name' )
152:       UNTIL done
153: END.

```

Number Play-by-Play

Without its main procedure, **NumberLines** (101–124), **Number** is like a shell—an unfinished program that handles various common details but lets you decide what the main purpose is to be. You could take out the **NumberLines** procedure and use the remaining shell as the basis for other programs that read text, do something to its lines, and write the result to another file.

The program begins, as do all Pascal programs, by identifying itself on line 1. (If you use the Output directive {*\$O name*} to send the compiled code to a file other than the program name, **Number**, put it above line 1.) Several comments (3–9) list the purpose of the program, what system it runs on, and the author. The compiler ignores everything between the symbols (* and *).

Lines 12–20 make up the program's global declarations. First is a new data type, **String64**, a string of 64 characters, large enough to hold most file and folder names. The value in brackets indicates how many characters the string can hold.

Variables **inFile** and **outFile** are both of type **TEXT**, a standard file type in Turbo as well as in other Pascal systems. Using standard file variables this way makes reading and writing text files on the Macintosh much easier than the typical approach of using the toolbox file manager, where you have to be concerned with such details as memory buffers, file markers, and other parameters. (The toolbox is the collection of programming and data types installed in ROM and in the System folder in every Macintosh.) There are times when these requirements give you more freedom, helping you to write better programs but, for short utilities such as **Number**, using standard **TEXT** files is as capable as other methods.

One problem when using **TEXT** files occurs if you read text created by other programs. For example, MacWrite and other word processors divide text into paragraphs, placing a carriage return at the end of each division. Because you normally use **Readln** statements to read lines out of a standard **TEXT** file, you'll discover that you cannot read paragraphs longer than the longest possible string—255 characters.

The reason for this problem is important to understand. Text lines and paragraphs exist only by agreement among the programs that read and write them. If you process a file that another program divides into paragraphs, Number thinks that the file's paragraphs are actually lines and tries to read them that way. It misinterprets the data in the file, an error that might cause the program to fail.

Is this a bug or not? It's probably more of a design limitation—something to be aware of when writing your own text file programs. You'll never have the problem reading text that you create with the Turbo editor, though. It ends all lines with carriage returns—the format that Number expects.

Function **Verified** (23–35) is worth extracting for your program library. It displays a message and asks you to type Y (Yes) or N (No), returning a Boolean value **TRUE** only if you type Y. Use the function this way:

```
IF Verified( 'Do you want to quit' )
  THEN Halt;
```

This displays the message “Do you want to quit? (y/n)” and ends the program with a **Halt** statement only if you answer yes. Notice that the function adds the question mark and (y/n) for you. This saves you from typing those characters at the end of every such yes-no prompt.

The way **Verified** works is to wait for you to type a single character and press the Return key at the **Readln** statement in line 33. If you type anything other than Y or y, line 34 sets the function to **FALSE** and ends. Otherwise, it returns **TRUE** before going back to the place in the program that called the routine.

If you would rather not press Return after typing a response, replace line 33 with these statements:

```
REPEAT
  ch := ReadChar
UNTIL ch IN [ 'Y', 'y', 'N', 'n' ];
Writeln( ch );
```

Turbo's **ReadChar** function waits for you to type a single character and, while it waits, displays a flashing underline cursor. As soon as you type any key, it returns that character as the function result, which this example then assigns to variable **ch**.

ReadChar does not display the character you type. For that reason, you might want to follow it with a **Writeln** statement so people can see their responses. It's

disconcerting to type something on a computer terminal but not be able to see what you type.

The **REPEAT** loop in this example shows how to limit responses to one of the four characters listed in brackets, an example of a *character set*. Only if the character in variable **ch** is **IN** the set of characters does the **REPEAT** loop end. If you type another character, it simply repeats again. Because **ReadChar** doesn't display anything, typing illegal responses appears to have no effect.

The second function in Number, **Duplicated** (38–63), checks whether a file already exists on disk. It returns **TRUE** only if it cannot find the file name you pass to the function. If it does find that same file, it returns **TRUE** only if you then give your permission to remove it. At lines 85–92 is an example showing how to use **Duplicated** to prevent programs from accidentally erasing existing files.

Notice that lines 47–49 set **Duplicated** to **FALSE** if they find a colon at the end of the file name. The **Pos** procedure returns the position of ':' (or any other character or string) in the second string, in this case, **name**. If that position equals the length of the string, then there's a colon at the end. This indicates that the name is a folder or volume and not a file.

Lines 52–62 check whether a valid file name exists. First, line 52 resets (opens) the file for reading and writing. But, in this case, the program doesn't actually read or write anything. It just wants to test whether it is *possible* to reset the file. If so, then the file exists. Otherwise it doesn't.

To accomplish this, line 52 surrounds the **Reset** statement with the compiler directives **{Si-}** and **{Si+}**, which turn off automatic Input/Output (I/O) error checking (–) or turn it back on (+). If it didn't do this and the file did not exist, the program would halt with an error. But with the directives in effect, it ignores errors, checking on its own for problems and taking appropriate action. The next **IF** statement does this by checking the integer value of the built-in **IoResult** function (53). With I/O error checking turned off, **IoResult** returns the result of the preceding I/O statement, in this example, the **Reset** instruction in the line above.

If **IoResult** was not zero, then there was an error trying to **Reset** the file, indicating that in all probability the file does not exist. (It's possible that a bad disk could cause an error here, but the program ignores that unlikely possibility.) If **IoResult** was zero, then the file exists and the program goes on to request permission to remove it. In that case, line 59 calls **Verified** for your yes-no response to the question, "Remove old name?" Notice how a **Concat** function—which joins two or more strings—passes that question along with the file name to **Verified**. Because of the **Concat**, if the file name is **Myfile.Txt**, the complete prompt comes out looking like this:

```
Remove old Myfile.Txt? (y/n)
```

The program assigns **Verified**'s result to the function identifier **Duplicated**, pass-

ing that result back to the program statement that called it. If you type Y, **Duplicated** returns **TRUE**; otherwise it returns **FALSE**.

Line 60, which closes the temporary file variable, is an optional step—Turbo automatically closes files at the ends of procedures in which you declare file variables. But it's probably a good idea to close local files anyway. A future change in the operating system or in the Turbo compiler might remove this invisible guarantee of an automatic close at procedure ends.

The third function in Number, **FilesOpened** (66–98), returns **TRUE** if it can open both the input and output files. This is a useful routine in any program that needs to open two files—one for reading and one for writing. If the function returns **TRUE**, you know it's okay to use the global **inFile** and **outFile** **TEXT** variables. If **FALSE**, then something went wrong opening a file or creating a new one. Also, if it returns **FALSE**, variable **done** indicates whether someone typed a name for the input file or merely pressed Return.

FilesOpened prompts for an input file name (77–79), gets your response, and sets **done** to **TRUE** if the length of that response is zero. The **IF** statement (80–96) takes effect only if **done** is **FALSE**, indicating that **inName** is not empty.

Line 82 uses the same method as **Duplicated** to check if file **inName** exists. Only if **IoResult** is zero does it go on to request an output file name. Continuing on, the program prompts for a name (85) and reads your response (86), checking here too (87) whether you press Return. Only if you didn't and only if function **Duplicated** returns **FALSE** at line 88, does the program attempt to create a new output file with a **Rewrite** statement (90). Notice the compiler directives there and the subsequent check of **IoResult**. If an error occurs when rewriting the file, this sets **okayFlag** **FALSE** and, therefore, returns **FALSE** when this same value passes back at line 97 as the function value.

All of this may seem overly involved merely to open two files for reading and writing. But the steps, while not inviolate, cover all possible situations and errors that might occur. Prove that claim to yourself by manually running through each program step and writing down the values of variables **okayFlag** and **done** for various scenarios. Put your own routines through similar *walk-throughs* rather than blindly trusting your impeccable programming skills without testing your inventions. You'll be surprised at the number of bugs you catch this way.

Up to this point, the program has ignored its main purpose, namely to add line numbers to text files. The procedure at lines 101–124 does this while illustrating how to read text files line-by-line. As I mentioned earlier, you can take this procedure out and store the result as a shell into which you can later insert various processes. Just replace **NumberLines** with your own routine.

NumberLines has a strange looking constant at line 107. The number sign (#)—some people call it a *pound sign*—specifies an ASCII (American Standard Code for Information Interchange) character of a specific value, in this case, an 8. Writing the backspace character to Turbo's dumb terminal display moves the cursor one

space to the left. Use the number sign to assign other control characters to character constants such as these:

```
CONST
```

```
    Return = #13; { Carriage return }
    Esc = #27;    { Escape character }
    CtrlX = #24;  { Control X }
```

Control characters are unusual on the Macintosh, which doesn't have the control key that other terminal keyboards typically have. To simulate control keys, use the **ReadChar** function explained earlier. With **ReadChar**, typing the Macintosh Command key and any letter returns a control character with the value of that letter's alphabetical position. In other words, Command-A returns ASCII 01, Command-B returns 02, and so on.

Most of **NumberLines**'s actions occur in the **WHILE** statement at lines 116–122. This loop repeats as long as the standard function **EOF** (End of File) returns **FALSE**. Therefore, the statements at lines 118–121 repeat until the program reaches the last line of text in **inFile**.

Line 119 reads one line of text from **inFile**, inserting that text into string variable **oneLine**, which is large enough to hold 255 characters, the maximum string length. The next line, 120, writes that same line to **outFile**, adding the line number (**lineNo**), a colon, and a blank. The **:3** after **lineNo** means “write the number in at least three columns.” This *format specification* lines up the numbers in neat columns. If your files have more than 999 lines, change 3 to 4.

Notice too that **Readln** (119) and **Writeln** (120) specify the input and output file variables as the first parameter in parentheses. This redirects the input and output through those variables—and, consequently, to the physical disk files. With no file variable parameters, **Readln** and **Writeln** (and also **Read** and **Write**) operate on the dumb terminal display.

The **Write** statement at line 121 uses the backspace character constant **bs** to move the cursor left four times after displaying the line number, placing it back at the extreme left edge of the window. Displaying line numbers this way gives you some feedback that the program is working. It's always a good idea to reassure people this way, letting them know that the program hasn't taken a left turn somewhere and is headed out into the countryside.

Procedure **DisplayInstructions** at lines 127–137 requires no special explanations. It merely displays a few reminders about how the program works. The main program body is at lines 140–153. This, too, is simple to understand. After calling **DisplayInstructions**, it repeats until function **FilesOpened** sets variable **done** to **TRUE**, indicating you pressed Return in response to the input file name prompt. In that event, the program calls **NumberLines** and then closes both the input and output files. If **FilesOpened** returns **FALSE**, the program displays an error message (150–151).

An improvement you can make is to move the two **Close** statements (147–148) to a separate procedure or function. You could turn off I/O error checking with the {*Si*–} compiler directive, and test **IoResult** after each **Close** to see whether there were any problems. Sometimes, it's possible to write a file to disk, but discover later that you cannot update the disk directory—the main responsibility of **Close**. The way the program stands now, it ignores this kind of error.

TABBING IN TEXT

The next two examples, *DeTab* and *ReTab*, operate similarly. Therefore, to cut down on duplication, they both use the text in Listing 2.2. Type it in and save as *TABS.INC* (meaning *IN*clude file). Turbo *includes* this text while compiling both programs.

Using include files to write programs cuts down on file sizes while building libraries of common routines that many programs share. This is the most basic way to reduce duplication among programs. In future chapters, you'll learn another way to do the same thing by writing your own precompiled library units. The difference between units and include files is that, by including common routines, you compile the procedures, functions, and other declarations for *each* program that uses those items. With units, you compile such elements only once. Even so, knowing how to use include files is important. Not only does it let you read common routines from a subroutine library, it also helps you divide large programs into pieces, letting the compiler join those pieces to produce a finished result.

Some of Listing 2.2 resembles the programming in *Number*. But don't reuse the procedures with the same names—they contain subtle but important differences.

Listing 2.2. TABS.INC

```

1: { Common declarations and routines for DeTab.PAS and ReTab.PAS }
2:
3:
4: CONST
5:
6:     TempName = 'TEMP.$$$';           { Temporary output file name }
7:     FixedTab = 4;                    { Default fixed tab width }
8:
9:
10: TYPE
11:
12:     String64 = String[64];
13:     String255 = String[255];
14:
15:
16: VAR
17:
18:     inFile, outFile : TEXT;           { Input, output files }
19:     inName, outName : String64;       { Input, output file names }
20:     done : BOOLEAN;                  { TRUE when no more files to do }
21:     tabWidth : INTEGER;              { Fixed tab width }
22:

```

(continued)

32 Programming with Macintosh Turbo Pascal

```
23:
24: PROCEDURE SeparateName( name : String64; VAR folder, fileName : String64 );
25:
26: { Separate name into two parts: a folder and file name }
27:
28:   VAR
29:
30:     p : INTEGER;
31:
32:
33:   FUNCTION ColonPosition( VAR p : INTEGER ) : INTEGER;
34:
35:   { Return indexed position of ':' in fileName, or return 0 }
36:
37:   BEGIN
38:     p := Pos( ':', fileName );
39:     ColonPosition := p
40:   END; { ColonPosition }
41:
42:
43: BEGIN { SeparateName }
44:   folder := '';           { Null string -- no space between quotes }
45:   fileName := name;
46:   WHILE ColonPosition( p ) > 0 DO
47:   BEGIN
48:     folder := CONCAT( folder, Copy( fileName, 1, p ) );
49:     Delete( fileName, 1, p )
50:   END { while }
51: END; { SeparateName }
52:
53:
54: FUNCTION FilesOpened( VAR done : BOOLEAN ) : BOOLEAN;
55:
56: { Return true if input and output files opened. }
57: { Set done to TRUE if no input file name entered. }
58:
59:   VAR
60:
61:     okayFlag : BOOLEAN;
62:     folder, fileName : String64;
63:
64:   BEGIN
65:     okayFlag := FALSE;    { assume the worst case }
66:     Write( 'Input file? ' );
67:     Readln( inName );
68:     done := length( inName ) = 0;
69:     IF NOT done THEN
70:     BEGIN
71:       {$i-} Reset( inFile, inName ); {$i+}
72:       IF IoResult <> 0
73:       THEN
74:         Writeln( 'ERROR: cannot find ', inName )
75:       ELSE
76:         BEGIN
77:           SeparateName( inName, folder, fileName );
78:           outName := concat( folder, TempName );
79:           {$i-} Rewrite( outFile, outName ); {$i+}
80:           okayFlag := ( IoResult = 0 );
81:           IF NOT okayFlag THEN
82:           BEGIN
83:             close( inFile );
84:             Writeln( 'ERROR: cannot create ', outName )
85:           END { if }
86:         END { else }
87:       END; { if }
88:       FilesOpened := okayFlag
89:     END; { FilesOpened }
90:
```

```

91:
92: PROCEDURE Report( oldLength, newLength : INTEGER );
93:
94: { Display a little report on before and after file sizes }
95:
96:   VAR
97:
98:       difference : INTEGER;
99:
100: BEGIN
101:   Writeln;
102:   Writeln( 'Original file length = ', oldLength:5 );
103:   Writeln( 'New file length      = ', newLength:5 );
104:   difference := abs( newLength - oldLength );
105:   IF newLength > oldLength
106:   THEN Writeln( 'Characters added      = ', difference:5 )
107:   ELSE IF newLength < oldLength
108:   THEN Writeln( 'Characters saved     = ', difference:5 )
109:   ELSE Writeln( 'No change in file length' );
110:   Writeln;
111: END; { Report }
112:
113:
114: PROCEDURE RenameFiles;
115:
116: { After all processing is done, rename original file <name>+Backup }
117: { and name the TEMP.$$$ output file to <name>. }
118:
119:   VAR
120:
121:       backupName : String64;
122:
123: BEGIN
124:
125:   backupName := concat( inName, ' Backup' );
126:
127:   {$i-} erase( backupName ); {$i+} { Erase old backup }
128:   IF IoResult <> 0
129:   THEN { ignore the error };      { ...don't worry if not there. }
130:   Rename( inName, backupName );   { Save copy of original file. }
131:   Rename( outName, inName )       { Rename new file. }
132: END; { RenameFiles }
133:
134:
135: PROCEDURE GetTabWidth;
136:
137: { Prompt for and set global tabWidth variable }
138:
139: BEGIN
140:   Writeln( 'Tab width is ', FixedTab );
141:   Writeln;
142:   Write( 'Enter new width, or 0 for no change : ' );
143:   Readln( tabWidth );
144:   IF tabWidth <= 0
145:   THEN tabWidth := FixedTab;
146:   Writeln
147: END; { GetTabWidth }

```

TABS.INC Play-by-Play

Because it's not a complete program, the listing does not start with **PROGRAM**. This is not an error. You might want to change the two constants at lines 6 and 7. The first, **TempName**, is the name of a temporary disk file for storing intermediate data. If you change it, be careful to choose a name such as TEMP.\$\$\$ that no other file is likely to use. Although the second constant, **FixedTab**, is 4, you could use 8 or 6 or any other small value for the default tab width. (Don't be too concerned with choosing the correct value—both ReTab and DeTab let you type another tab width if you don't want to use the default value.)

Two string constants **String64** and **String255** define string types of 64 and 255 characters (12–13). As in **Number**, **inFile** and **outFile** (18) are standard **TEXT** files while **inName** and **outName** (19) are of the 64-character string type. Boolean variable **done** controls when the program ends, and **tabWidth** equals the number of spaces in each tab column, using the default **FixedTab** value if you don't specify another.

The first procedure in TABS.INC is **SeparateName** (24–51), a tool for which you'll undoubtedly find other uses. It takes as its parameter a file name and returns in the two variables, **folder** and **fileName**. If you set **name** to Programs: Textbook: Number.PAS and call **SeparateName**, it returns **folder** equal to the string 'Programs: Textbook:' and **fileName** equal to 'Number.PAS.'

Separating file names into their components gives you the option of accessing other files in the same folder without typing the folder name over and over. It also lets DeTab and ReTab perform a critical operation, one that many word processors and text editors—including the Turbo editor—fail to do: save a backup copy of your text. The next section explains the technique.

Backing Up Files

With a backup copy of your text containing all lines as they existed before processing, you don't have to ask "Remove old Data.TXT?" as **Number** does. Saving a backup copy is a better idea because it avoids the obvious problem that often occurs when people answer Yes to that question when they mean No. In general, to save a backup copy requires these seven steps:

1. Open the input file
2. Create a temporary output file
3. Process the lines in the input file, writing the new lines to the output
4. If an error occurs, erase the temporary file and end. Otherwise, continue with step 5
5. Erase any old backup file
6. Rename the original file "<name> Backup"
7. Rename the temporary file "<name>"

Programs that follow this plan protect you from accidentally erasing files. Even if you take all the tabs out of your only copy of an important report, you can recover your original text by throwing away the new file and using the backup. This is an important feature to add to all programs that create files on disk.

Function **FilesOpened** (54–89) is similar to the same routine in *Number* but, because of the backup copy scheme, it no longer asks whether you want to remove an existing file. For that reason, it's a bit more useful than the other version and you might want to use it instead. If you followed the comments about the earlier program, you shouldn't have any trouble understanding how this one works.

Procedure **Report** (92–111) displays a few statistics about the before and after file lengths. When changing blanks to tabs with *ReTab*, it's nice to know how much disk space you saved. Because of the heavy indentation in Pascal programs, you might be surprised to learn that you can often save quite a lot of space—as much as 40% or more per file—just by scrunching blanks into tabs.

The next procedure, **RenameFiles**, (114–132), performs steps 5, 6, and 7 from the plan outlined earlier. First, it adds “Backup” to the end of the input file name (125) and then erases any file now on disk with that name (127). Removing the old backup file is a most important step. If you don't do it, the following **Rename** procedures will not work properly.

Notice that lines 128–129 ignore any errors reported by **IoResult** from the erase operation in the previous line. This takes care of the good possibility that there won't be any old backup file on disk. At the same time, it demonstrates an important requirement when turning off I/O error checking with {*Si*–} as we've been doing. When you use this method, you *must* check **IoResult** following all I/O operations—even if, as in this case, you plan to ignore any errors. The reason for this requirement is that if any errors do occur, they set an invisible flag inside Turbo Pascal's run-time routines, which contain the code for various commands and features. When errors occur with I/O checking off, this internal flag prevents future I/O operations until you check **IoResult**. If you don't check it, then, errors can shut down all future I/O, meaning that **Readln**, **Writeln**, and other file operations won't work from then on.

RenameFiles ends by renaming the input file to **backupName**, which saves the backup copy of the original text. It also renames the output file—which now has the name you assign to the constant **TempName**.

The final procedure, **Get TabWidth** (135–147), lets you type a different value than the default constant **FixedTab**. Both *DeTab* and *ReTab* call this procedure to let you change tab widths.

REMOVING TABS FROM TEXT

DeTab is the simpler of the two tab utilities. Type in Listing 2.3 and save as *DETAB.PAS*. You can compile it to memory or to disk. If you have trouble compiling, you might have to change the include directive in line 12. On my system,

I name my disk volume Programs and store DETAB.PAS, RETAB.PAS, and TABS.INC in the folder Textbook. Line 12 includes TABS.INC, compiling the common routines and data. If you call your disk and folders something else, insert their names here.

When you run DeTab, supply the name of the file from which you want to remove tab characters, replacing them with blanks while maintaining the same column spacing in the original. Be aware that, because it replaces single tabs with multiple blanks, the program usually increases the file size. It could easily double or triple the size of a heavily tabbed file. Make sure you have plenty of free space on disk.

Unless it detects errors, DeTab stores the result in a new file with the same name as your original. Your old text—with its tabs intact—is in the backup file (with the name of the original plus “Backup”).

Listing 2.3. DETAB.PAS

```

1: PROGRAM DeTab;
2:
3: (*
4:
5:  * PURPOSE : Remove tabs from a text file
6:  * SYSTEM  : Macintosh / Turbo Pascal / Textbook interface
7:  * AUTHOR   : Tom Swan
8:
9:  *)
10:
11:
12: {$I Programs:Textbook:Tabs.INC }
13:
14:
15: PROCEDURE ProcessLine( VAR line : String255 );
16:
17: { Remove tabs from this line. }
18:
19:   CONST
20:
21:     Blank = ' ';           { One blank character }
22:     Tab   = ^I;           { Control-I = ASCII tab character }
23:
24:   VAR
25:
26:     temp : String255;      { Temporary string holder }
27:     i    : INTEGER;       { FOR-loop control variable }
28:
29:   BEGIN
30:     temp := '';           { Null string -- no space between the quotes }
31:     FOR i := 1 TO Length( line ) DO
32:       BEGIN
33:         IF line[i] = Tab
34:           THEN
35:             REPEAT
36:               temp := concat( temp, Blank )
37:             UNTIL ( Length( temp ) MOD tabWidth = 0 )
38:           ELSE
39:             temp := concat( temp, line[i] )
40:           END; { for }
41:         line := temp
42:       END; { ProcessLine }
43:

```

```

44:
45: PROCEDURE ProcessFile;
46:
47: { Read lines from inFile, remove tabs, and write to outFile }
48:
49:   CONST
50:
51:       bs = #8;      { Backspace control char }
52:
53:   VAR
54:
55:       oneLine      : String255;      { Holds one line of text }
56:       lineNo       : INTEGER;        { For displaying line numbers }
57:       oldLength,
58:       newLength    : INTEGER;        { Statistics }
59:
60:   BEGIN
61:       lineNo := 0; oldLength := 0; newLength := 0;
62:       WHILE NOT EOF( inFile ) DO
63:           BEGIN
64:               lineNo := lineNo + 1;
65:               Readln( inFile, oneLine );
66:               oldLength := oldLength + Length( oneLine ); { Length before }
67:               ProcessLine( oneLine );
68:               newLength := newLength + Length( oneLine ); { ...and after }
69:               Writeln( outFile, oneLine );
70:               Write( lineNo:4, bs, bs, bs, bs ) { display line numbers }
71:           END; { while }
72:           Writeln( lineNo:4, ' lines done' );
73:           Report( oldLength, newLength )
74:       END; { ProcessFile }
75:
76:
77: PROCEDURE Initialize;
78:
79:   BEGIN
80:       Writeln( 'DETAB: Remove tabs from a text file' );
81:       Writeln;
82:       Writeln( '1. For the input file, type the name of any text file.' );
83:       Writeln( '2. The program removes tabs from the text, replacing' );
84:       Writeln( '   them with the correct number of blanks.' );
85:       Writeln( '3. Use the program to convert tabbed text files created' );
86:       Writeln( '   with a text editor such as the one in the MDS 68000' );
87:       Writeln( '   development system for use by Turbo Pascal.' );
88:       Writeln( '4. The program saves a copy of your original text,' );
89:       Writeln( '   adding "Backup" to the end of the file name.' );
90:       Writeln;
91:       GetTabWidth
92:   END; { Initialize }
93:
94:
95: BEGIN
96:   Initialize;
97:   REPEAT
98:       Writeln;
99:       IF FilesOpened( done ) THEN
100:           BEGIN
101:               ProcessFile;
102:               Close( inFile );
103:               Close( outFile );
104:               RenameFiles
105:           END
106:       UNTIL done
107: END.

```

DeTab Play-by-Play

DeTab's main action occurs in procedure **ProcessLine** (15–42). The program sends each line of text to **ProcessLine**, which removes the tabs it finds, replacing them with blanks. It does this by cycling through each character in a **FOR** loop at lines 31–40. If the character is a tab (33), then the loop adds a blank to the end of a temporary string with a **Concat** statement (36). It repeats this until the length of the string equals some multiple of the **tabWidth**, a calculation it performs at line 37. If the character is not a tab, line 39 adds it to the end of the temporary string.

After examining each character, **ProcessLine** ends at line 41 by reassigning the temporary string back to the **line** parameter, passing the now tabless string back.

ProcessFile (45–74) and **Initialize** (77–92) are nearly identical to **NumberLines** and **DisplayInstructions** in program Number (Listing 2.1). **ProcessFile** keeps track of the old and new line lengths for the later report on file size savings. **Initialize** displays instructions and ends with a call to **GetTabWidth** (91) to let you enter a different tab value each time you run the program. If you will always use the default setting, remove line 91.

The main loop is simpler than it is in Number because **FilesOpened** now handles its own error messages. Notice that line 104 calls **RenameFiles** after closing the input and output files. This is the step that renames the temporary output file TEMP.\$\$\$ and saves your original text by adding Backup to its name.

ADDING TABS TO TEXT

ReTab runs similarly to DeTab but does the opposite job. It adds tabs to text files, replacing as many multiple blanks as possible with tab characters. This can reduce the size of text files, especially Pascal listings which typically have many blanks in front of indented lines.

You can use ReTab to save *archival* copies of source code listings that you don't want to discard but are not likely to need soon. To bring a file out of the archives and revive it for further editing, run it through DeTab, replacing the tabs with blanks.

Type in Listing 2.4 and save as RETAB.PAS. Compile the program either to memory or to a disk code file. When you run it, type the name of the text file to which you want to add tabs. The program reads your text, converts multiple blanks to tabs and saves a copy of your file as a backup.

Listing 2.4. RETAB.PAS

```

1: PROGRAM ReTab;
2:
3: (*
4:
5:  * PURPOSE : Add tabs to a text file
6:  * SYSTEM  : Macintosh / Turbo Pascal / Textbook interface
7:  * AUTHOR   : Tom Swan
8:
9:  *)
10:
```

```

11:
12: {$I Programs:Textbook:Tabs.INC }
13:
14:
15: PROCEDURE ProcessLine( VAR line : String255 );
16:
17: { Add tabs to this line. }
18:
19: {
20:   Based on an algorithm from:
21:   "Software Tools in Pascal"
22:   by Brian W. Kernighan and P. J. Plauger
23:   Addison-Wesley Publishing Company; 1981
24: }
25:
26:   CONST
27:
28:       Blank = ' ';           { One blank character }
29:       Tab   = ^I;           { Control-I = ASCII tab character }
30:       cr    = ^M;           { Control-M = ASCII cr character }
31:
32:   VAR
33:
34:       temp : String255;      { Temporary string holder }
35:       col, newCol : INTEGER; { Line indexes }
36:       ch : CHAR;             { Single character holder }
37:
38:
39:   FUNCTION NextChar( VAR ch : CHAR ) : CHAR;
40:
41:   { Return next character from line }
42:
43:   BEGIN
44:       ch := line[ succ( newCol ) ];
45:       NextChar := ch
46:   END; { NextChar }
47:
48:   BEGIN
49:       temp := '';           { Null string -- no space between the quotes }
50:       line := concat( line, cr ); { Add end of line marker to line }
51:       col := 0;
52:       REPEAT
53:           newCol := col;
54:           WHILE NextChar( ch ) = Blank DO
55:               BEGIN
56:                   newCol := newCol + 1;
57:                   IF newCol MOD TabWidth = 0 THEN
58:                       BEGIN
59:                           temp := concat( temp, Tab );
60:                           col := newCol
61:                       END { if }
62:                   END; { while }
63:                   WHILE( col < newCol ) DO
64:                       BEGIN
65:                           temp := concat( temp, Blank );
66:                           col := col + 1
67:                       END; { while }
68:                   IF ch <> cr THEN
69:                       BEGIN
70:                           temp := concat( temp, ch );
71:                           col := col + 1
72:                       END { if }
73:                   UNTIL ch = cr;
74:                   line := temp
75:               END; { ProcessLine }
76:
77:

```

(continued)


```

78: PROCEDURE ProcessFile;
79:
80: { Read lines from inFile, add tabs, and write to outFile }
81:
82:   CONST
83:
84:       bs = #8;      { Backspace control char }
85:
86:   VAR
87:
88:       oneLine      : String255;      { Holds one line of text }
89:       lineNo       : INTEGER;        { For displaying line numbers }
90:       oldLength,
91:       newLength    : INTEGER;        { Statistics }
92:
93:   BEGIN
94:       lineNo := 0; oldLength := 0; newLength := 0;
95:       WHILE NOT EOF( inFile ) DO
96:           BEGIN
97:               lineNo := lineNo + 1;
98:               Readln( inFile, oneLine );
99:               oldLength := oldLength + Length( oneLine ); { Length before }
100:              ProcessLine( oneLine );
101:              newLength := newLength + Length( oneLine ); { ...and after }
102:              Writeln( outFile, oneLine );
103:              Write( lineNo:4, bs, bs, bs, bs ) { display line numbers }
104:           END; { while }
105:           Writeln( lineNo:4, ' lines done' );
106:           Report( oldLength, newLength )
107:       END; { ProcessFile }
108:
109:
110: PROCEDURE Initialize;
111:
112:   BEGIN
113:       Writeln( 'RETAB: Add tabs to a text file' );
114:       Writeln;
115:       Writeln( '1. For the input file, type the name of any text file.' );
116:       Writeln( '2. The program adds tabs to the text, replacing groups' );
117:       Writeln( '   of blanks with tabs wherever possible.' );
118:       Writeln( '3. After converting, the text may take less disk room,' );
119:       Writeln( '   but may look "strange" in the Turbo Pascal editor.' );
120:       Writeln( '4. The program saves a copy of your original text' );
121:       Writeln( '   adding "Backup" to the end of the file name.' );
122:       Writeln;
123:       GetTabWidth
124:   END; { Initialize }
125:
126:
127: BEGIN
128:   Initialize;
129:   REPEAT
130:       Writeln;
131:       IF FilesOpened( done ) THEN
132:           BEGIN
133:               ProcessFile;
134:               Close( inFile );
135:               Close( outFile );
136:               RenameFiles
137:           END
138:       UNTIL done
139:   END.

```

ReTab Play-by-Play

Except for procedure **ProcessLine** (15–75), all of ReTab is similar to DeTab. The procedure follows an algorithm for replacing sequences of blanks with tabs while maintaining the same relative position of columns.

The main action occurs in a **REPEAT** statement at lines 52–73 that examines each character in a line with the help of function **NextChar** (39–46). Inside the **REPEAT** loop, a **WHILE** loop (54–62) tests successive characters, adding tabs in place of blanks whenever variable **newCol** reaches a fixed tab position.

A second **WHILE** loop (63–67) adds blanks to the line to fill out columns with less than the minimum number of blanks that a tab could replace. This situation occurs only if variable **col** is less than **newCol**. (The prior **WHILE** loop assigns **newCol** to **col** every time it inserts a tab into the line.) Together, the two **WHILE** loops compress as many blanks as possible into tabs.

Following that, an **IF** statement (68–72) adds non-blank characters to the line and checks for a carriage return (cr) which ends the **REPEAT** loop at line 73. This works because line 50 adds a carriage return to the end of the line. It has to do this because, when **ReadLn** reads strings, it never ends them with carriage return characters. You could use a different ending character as a flag but, because you can be sure strings never will have carriage returns in them, this seems to be the best choice.

Together, DeTab and ReTab add and remove tabs from text files. You can use DeTab to remove tab characters from files that another text editor or word processor created. This is necessary because, even though it lets you press the tab key, the Turbo editor doesn't actually insert tab characters into text. Instead, when you tab, Turbo adds the correct number of spaces at that point, simulating what a tab control character usually does. Even worse, when Turbo finds tabs in text, instead of lining up characters into neat columns, it simply ignores them! If text files from other editors look odd in Turbo, try processing them with DeTab.

Use the other program, ReTab, to add tabs to text files, replacing multiple spaces with tab characters without changing the column spacing. This can reduce the amount of room that a file takes on disk by compressing multiple spaces into single tab characters.

Number, DeTab, and ReTab are three useful utilities, all of a category of programs that read and write text files. You should be able to use many of the ideas in them in your own textbook programs. To convert other kinds of programs from IBM PC Turbo Pascal to the Macintosh, though, requires more information about the differences between the two systems, as the next section explains.

CONVERTING IBM PC PROGRAMS TO MACINTOSH

The following list of identifiers either do not exist in Macintosh Turbo Pascal or differ from the IBM PC and CP/M versions. These notes help you to convert programs from one system to another.

The first section describes identifiers that changed or disappeared in the Macintosh compiler. The second section describes new identifiers found only on the Macintosh version. Use the first section to convert IBM PC programs to the Macintosh. Use the second to avoid writing Macintosh programs that will be difficult to convert later to the IBM PC.

The following is not a complete reference to Turbo Pascal. For that, and for the exact format of these and other commands, consult the *Guide* and other references. Some of these commands apply only to textbook style programs or to those that use PasInOut, PasConsole and other units. Some of the programming references in the notes require toolbox interfaces such as MemTypes, QuickDraw, OSIntf, ToolIntf, and others.

IBM PC Identifiers Changed or Deleted

Addr

Use *@* in front of the identifier whose address you want. For example, instead of **Addr(wordsArray)**, write **@wordsArray**.

Append

No equivalent.

Assign

This is a major change. In IBM PC Turbo Pascal, you assign a file name to a file variable before resetting or rewriting that file. There's no equivalent procedure in Macintosh Turbo Pascal. Instead, now there are two forms of **Reset** (see later description). Use the original form **Reset(f)** to rewind a file to its top, not to open it the first time. Instead, use **Reset(f,name)** to open existing files. This takes the place of the two IBM PC statements:

```
Assign( f, name );
Reset( f );
```

BDOS, BDOSHL, BIOS, and BIOSHL

No equivalents. **BDOS** and **BIOS** commands call routines in the CP/M operating system for that version of Turbo Pascal. On the Macintosh, you do similar jobs by calling toolbox procedures.

BlockRead and BlockWrite

No equivalents. These two procedures read and write disk blocks with no regard for their contents. You use them in programs that don't care what files contain. For example, a copy program might use them to copy the bytes in one file to another.

The Macintosh toolbox has its own such low-level disk routines, **FSRead** and **FSWrite**, which Volume 2 of *Inside Macintosh* describes how to use. These procedures are more powerful than **BlockRead** and **BlockWrite** because they read and write any number of bytes rather than only entire blocks at a time.

Chain

You cannot chain from one program to another. Use the {SS+} and {SS SegName} compiler directives to segment large programs into pieces, or overlays.

Another possibility is to add a Transfer menu to your program and let people choose other programs to run. (See Chapter 7.) Or, you could hard-wire the names of certain programs to which others transfer, in effect simulating a Chain.

ChDir

Use the toolbox File manager function **SetVol** to change the default volume number for subsequent disk operations.

ClrEol and ClrScr

These procedures exist but have the new names **ClearEol** and **ClearScreen**. They work only with dumb terminal textbook style programs.

Rather than revise all such commands in every IBM PC program you convert to the Macintosh, write two procedures named **ClrEol** and **ClrScr**. For example:

```
PROCEDURE ClrEol ;  
BEGIN  
    ClearEol  
END; { ClrEol }
```

This translates the IBM PC **ClrEol** to the new spelling, **ClearEol**, and avoids having to touch up your program's source code.

CrtExit and CrtInit

These two procedures send control codes (usually) to the terminal before (**CrtInit**) and after (**CrtExit**) programs run. There are no equivalents and no need for them on the Macintosh.

CSeg

No equivalent. This is a processor-dependent function that returns the Code segment of the 8088/86 processor. There is no similar value on 68000 systems like the Macintosh.

DeleteLine

The same procedure exists, but it has a new name, **DeleteLine**. It deletes the display line at the cursor position in Turbo's dumb terminal textbook window.

Delay

Although it's no longer one of Turbo's native commands, the Macintosh toolbox has a similar routine with the different form:

```
Delay( numTicks : LONGINT; VAR finalTicks : LONGINT );
```

NumTicks is the number of ticks, or heartbeats, in 1/60-second intervals to delay. **FinalTicks** is the number of such heartbeats from the time you turned on the Macintosh until the delay ends.

Don't trust this routine for complete accuracy. Use it only in situations where an approximate time delay is adequate.

Draw

Use Turbo's Turtlegraphics unit commands **PenDown**, **Forwd**, and **Back** (among others) to draw lines. Or, even better, use toolbox QuickDraw procedures **LineTo** and **Line**.

DSeg

No equivalent. This is a processor-dependent function that returns the Data segment of the 8088/86 processor. There is no similar value on 68000 systems like the Macintosh.

Erase

The same procedure exists for programs that use the **PasInOut** unit, as do all textbook style programs. But it has the new form:

```
Erase( fileName );
```

where **fileName** is a string. Other Turbo versions assign a name to a file variable and use **Erase(f)** to erase that file. The new form replaces those two steps.

Execute

No equivalent. See Chain.

Flush

No equivalent. In textbook programs, close and reopen files to flush in-memory data to disk. In fully charged Macintosh programs, call the File manager routine **FlushVol**.

Frac

To extract the fractional part of a real number, add this function to your program.

```
FUNCTION Frac ( r : REAL ) : REAL;
BEGIN
    Frac := r - INT(r)
END; { Frac }
```

FreeMem

No equivalent. Use toolbox memory manager routines to manipulate the heap.

GetDir

Use the File manager **GetVol** function to get a specific drive name.

GetMem

No equivalent. Use toolbox memory manager routines **NewHandle** and **NewPtr** to create memory areas in the heap.

GraphBackground and GraphColorMode

No equivalents. Use QuickDraw routines instead.

HighVideo

No equivalent. Textbook style programs use 9-point Monaco for text with no reversed video or other display attributes. To display various fonts and text styles in windows, use QuickDraw text routines.

Hires and HiresColor

On the IBM PC, these commands switch to the graphics display and set the drawing color. The Macintosh graphically draws *everything* and the commands aren't needed.

InLine

You can insert in-line, 68000 machine language into Pascal programs but not in the same way you insert 8088/86 machine code into IBM PC programs. On the Macintosh, in-line code follows a procedure header. Everywhere in your program that you use that procedure name, the compiler inserts the in-line code. This operates more like an assembler macro—a way to consolidate instructions under a single label—than a direct insert of machine language into what the compiler produces.

Converting IBM PC programs that heavily use **InLine** statements is extremely difficult. Probably, the best plan would be to remove the **InLine** statements from the PC program and, on that computer, write equivalent Pascal procedures. After the program is working correctly, translate it to Macintosh Turbo Pascal. Then, if certain operations need the extra speed that only machine language can give, convert them to 68000 in-line code. This method avoids the difficulties of trying to translate 8088/86 into 68000 machine language, a shaky limb upon which I do not wish to stand for very long.

InsLine

The same procedure exists, but has a new name, **InsertLine**. It inserts a blank display line at the cursor position in Turbo's dumb terminal textbook window.

Kbd

No equivalent. In IBM PC programs, this standard file typically goes in a **Read** statement such as this:

```
Read( Kbd, ch );
```

With this technique, you read a single character from the keyboard but don't display that character. This is useful when you want to avoid pressing Return after typing single character responses such as Y or N and also when you want to read control characters but not display them.

To do the same thing on the Macintosh, use the **ReadChar** function as explained early in this chapter.

LongFilePos, LongSeek, and LongFileSize

These three procedures let you use real numbers to access records at random in disk files. They eliminate the file size restrictions that integer record numbers impose. To do the same thing on the Macintosh, use the toolbox File manager routines **GetFPos** for **LongFilePos**; **SetFPos** for **LongSeek**; and **GetEOF** for **LongFileSize**. The toolbox routines use the **LONGINT** data type for byte counts and, therefore, increase file size to the maximum long integer of 2,147,483,647. That's

equal to more than 4 million 512-byte disk blocks—a goodly amount as they say in our part of the country.

LowVideo

No equivalent. See HighVideo.

Mark

Memory management is more sophisticated on the Macintosh than the simple ability provided by marking the heap and, after creating objects there, releasing that memory with the standard **Release** procedure.

Some people will lament the passing of **Mark** and **Release** from Macintosh Turbo Pascal. But you can get into serious trouble if you attempt to manage the heap in this simplistic way. Undoubtedly, your program would conflict with pull down menus, windows, controls, dialogs, desk accessories, and other processes that also use memory while your program runs.

After creating dynamic objects with **New**, use **Dispose** to recover their memory for later use. It's difficult to convert programs that rely on **Mark** and **Release** for memory management, but not impossible. Just be certain to **Dispose** every variable the program creates with **New**. This lets Turbo and the Macintosh toolbox manage memory in ways that won't conflict with other processes.

MkDir

No equivalent. Create your own folders with the Finder.

Move

Use **MoveLeft** and **MoveRight** instead, two procedures that move memory bytes to lower (**MoveLeft**) or to higher (**MoveRight**) addresses. The original **Move** prevented you from accidentally moving in the wrong direction and, in the process, destroying data you meant to preserve. The replacement procedures do allow this to happen but only if the destination and source areas overlap. If your source and destination areas do not overlap, then you can safely use either **MoveLeft** or **MoveRight**. In that case, they have identical effects.

MsDos

No equivalent.

NormVideo

No equivalent. See HighVideo.

NoSound

Use the toolbox Sound driver procedure **StopSound** instead.

Ofs

No equivalent. This is a processor-dependent function that returns an offset address for variables on systems using the 8088/86 processor.

Palette

No equivalent. Use the QuickDraw routine **PenPat** to change pen patterns and to draw in various shades of gray.

ParamCount and ParamStr

The toolbox segment loader routines **CountAppFiles**, **GetAppFiles**, **ClrAppFiles**, and **ClrAppParms** duplicate what these two IBM PC procedures do. Generally, you use **CountAppFiles** in place of **ParamCount** and **GetAppFiles** in place of **ParamStr**. But, unlike on the IBM PC, the Macintosh procedures return only the names of data files opened by clicking their icons in the Finder. Another time to use **GetAppFiles** is when someone opens several files along with an application at the same time.

The IBM PC version lets people type other parameters—not just file names—that programs can recover with **ParamCount** and **ParamStr**. In this sense, you cannot do the same thing on the Macintosh.

Plot

To plot a point with Turtle Graphics, use the following two statements:

```
PenDown;  
Forwd ( 0 );
```

This places a tiny dot at the current pen location. (You might need a magnifying glass to see it.) Another way to plot points at a location (h,v) is with these QuickDraw statements:

```
MoveTo( h, v );  
LineTo( h, v );
```

Ptr

The toolbox memory manager data type **Ptr** replaces this function, which sets a pointer variable to a specific address in CP/M and IBM PC Turbo Pascal systems.

You can do the same thing on the Macintosh by using **POINTER** to assign specific addresses to pointers. For example, this sets pointer p to address 3:

```
VAR
  p : ^INTEGER;
BEGIN
  p := POINTER( 3 )
END.
```

Ptr is now a data type declared in the **MemTypes** interface. You can simulate the way it works on the PC by declaring a variable of type **Ptr** and then using type coercion to assign specific values to that variable. (A coercion forces Pascal to treat data as a different data type than its original declaration.) You could write:

```
USES MemTypes;
VAR
  p : Ptr;
BEGIN
  p := Ptr( 3 )
END.
```

But notice that this is not quite the same thing as assigning an address to any type of pointer variable. In order for the coercion to work, the variable must be of type **Ptr**.

Random and Randomize

Use QuickDraw's **Random** function instead. Unfortunately, this returns only integer values, not reals in the range $0 \leq \text{random} < 1.0$. Use **RandSeed** in place of **Randomize** to start new random sequences.

Release

No equivalent. See **Mark**.

Rename

The same procedure exists for textbook programs, but it takes two string parameters instead of a file variable and string as in the IBM PC version. To rename fileNameA to fileNameB do this:

```
{ $i - }
Rename( fileNameA, fileNameB );
{ $i + }
IF IoResult <> 0
  THEN writeln( 'Error renaming!' );
```

Reset and Rewrite

For textbook programs that use Turbo's **PasInOut** interface, **Reset** and **Rewrite** are similar to their IBM PC counterparts. Both now come in two forms. To open a file for reading, use **Reset(f,name)**. To rewind a file to its top after opening, use **Reset(f)** without the name. To create a new file, use **Rewrite(f,name)**. To rewind a file and start writing it from the top, use **Rewrite(f)** without a name.

When you **Rewrite** a text file, you can only write to it. When you **Reset** a text file, you can only read from it. See also the comments under **Assign**.

Rmdir

No equivalent. Use the Finder to remove folders.

Seg

No equivalent. This is a processor-dependent function that returns a segment address for variables on systems using the 8088/86 processor.

Sound

Use the toolbox **Sound** driver procedure **StartSound** instead.

SSeg

No equivalent. This returns the stack segment register value on systems using the 8088/86 processor.

Str

Use the Binary-Decimal Conversion Package procedures **StringToNum** and **NumToString** to convert strings and integer values. Unfortunately, unlike **Str**, you cannot format strings as you can in statements such as **Str(r:8:2,s)** on the IBM PC.

TextBackground, TextColor, and TextMode

No equivalents. Use QuickDraw routines to change text fonts and styles in windows.

UpCase

No equivalent although you can use the OS (Operating System) Utility routine **UpString** to convert strings to uppercase. (**UpCase** converts only single characters.)

It's a simple matter to write your own **UpCase** function, duplicating the IBM PC command. Here's one way to do it:

```

FUNCTION UpCase( ch : CHAR ) : CHAR;
BEGIN
  IF ch IN [ 'a' .. 'z' ]
    THEN UpCase := CHR( Ord(ch) - 32 )
    ELSE UpCase := ch
END; {UpCase}

```

Val

No equivalent. Use toolbox procedures **StringToNum** and **NumToString** to convert strings and numbers. See also **Str**.

WhereX and WhereY

No equivalent. In Turbo's textbook window, it's not possible to locate the cursor position. But, because the Macintosh uses a graphics display for everything you see, you can always use QuickDraw routines such as **GetPen** to locate where text will appear. Remember, though, that this returns a single point on the Macintosh's 512×342 visible display (or in different limits on larger-screen models), not an (x,y) character coordinate as **WhereX** and **WhereY** do.

Window

Use the Window manager. You cannot use IBM PC pseudo-windows in Macintosh textbook style programs.

New Macintosh Identifiers Not in IBM PC Turbo Pascal

Ord4

This returns the long integer ordinal (whole number) value of 32-bit objects. Use **Ord4** to convert pointers to **LONGINT** variables. For example, if you have a pointer **p**, and a variable **Addr** of type **LONGINT**, you could write:

```
Addr := Ord4( p );
```

Pointer

This function replaces **Ptr** in the IBM PC version. Use it to assign specific addresses to pointer variables or to convert one pointer type to another. To assign a specific address, do this:

```

VAR
  p : ^INTEGER;
BEGIN
  p := POINTER( 3000 )
END.

```

To convert one pointer type to another, use **POINTER** in the assignment. Because the function returns a generic type, you can always assign it to any pointer variable. For example,

```
VAR
  recPtr : ^AnyRecord;
  intPtr : ^INTEGER;
BEGIN
  intPtr := POINTER( 3000 );
  recPtr := POINTER( intPtr )
END.
```

Doing this assigns the same address to both **intPtr** and **recPtr**. Use this technique if you need two or more pointers to the same memory location but want to interpret the contents of that memory as different objects.

Float

It's doubtful you'll ever use **Float**. It converts integer values to reals. If you have an integer **n**, you can assign it to a real variable **r** with the statement:

```
r := Float( n );
```

The only reason to do this is for documentation. In other words, **Float** makes it obvious that you are converting a variable from one type to another, a fact that in mathematically critical programs may be important to know. But you can always write more simple statements such as:

```
r := i;
```

to do the same thing. This works because Turbo converts integers to reals in expressions that it evaluates to real results. Remember that the reverse is not true: you cannot directly assign reals to integers. For that, use the standard functions **Trunc** or **Round**, the complements of **Float**.

ClearEol and ClearScreen

These two procedures replace **ClrEol** and **ClrScr** in IBM PC Turbo Pascal. They do the same thing—clear from the cursor to the end of the line and clear the entire screen—as the PC commands. They work only in textbook programs that use the **PasConsole** unit.

ReadChar

ReadChar reads single characters from the keyboard without displaying those characters in the dumb terminal textbook window. **ReadChar** simulates control

characters when you press the Command key and then type a character. It returns ASCII controls for keys Return (13); Enter (3); and Clear (27), as well as for others such as the arrow keys on a numeric keypad or on a Macintosh Plus keyboard.

MoveLeft and MoveRight

MoveLeft moves bytes in memory from higher to lower addresses. **MoveRight** does the same, but it moves from lower to higher addresses. On the IBM PC, the single **Move** does the same thing, figuring out for you the correct direction to move while avoiding overlapping bytes from the source to the destination.

In one sense, **Move** is more powerful than **MoveLeft** and **MoveRight**—it helps you avoid errors. But in another sense, the two commands are improvements. For one thing, because they don't have the logic that decides in which direction to go, you can assume that **MoveLeft** and **MoveRight** run faster than **Move** would if it existed in the Macintosh compiler.

Be extremely careful with these two instructions. They move memory with no regard for other items they might disturb. A good rule to follow is: if you move non-overlapping blocks, use either **MoveRight** or **MoveLeft**; if you move overlapping blocks, use **MoveRight** when the destination is higher than the source; use **MoveLeft** when the destination is lower.

ScanEQ and ScanNE

These two functions scan memory for matching (**ScanEQ**) or non-matching (**ScanNE**) byte values. To scan the first 100 bytes of an array **MyArray** for the offset byte position from the start of the array to the first occurrence of a byte 255, you could write:

```
Offset := ScanEQ( 100, 255, MyArray );
```

If **Offset** has the value 100, then **ScanEQ** did not find 255 in the array. If **Offset** is in the range 0 . . 99, it represents the position **MyArray[Offset]** where it found the byte value 255.

ScanNE works the same way, but it finds the occurrence of the first value not equal to another.

HiWord, LoWord, and SwapWord

These three new commands are the **LONGINT** equivalents of **Hi**, **Lo**, and **Swap**, available in all versions of Turbo Pascal. **HiWord** returns the high order 16-bit **INTEGER** value of a 32-bit **LONGINT** variable. **LoWord** returns the low order **INTEGER**. **SwapWord** reverses the low and high words of a 32-bit value.

When dealing with specific parts of integers, remember that byte values are already swapped on the IBM PC and CP/M Turbo Pascal versions. In fact, on most

8- and many 16-bit computers, the low byte (0 . . 255) is physically ahead (at a lower address) of the high byte ((0 . . 255)*256). In computers with 68000 processors, the low byte comes after (at a higher address) the high byte value.

Such machine dependencies quickly lead to migraines and other programmer maladies. In subjects such as these, the best medicine is preventive: don't write machine dependent code in the first place.

Turtle Graphics vs. QuickDraw

There are two ways to draw images with Turbo Pascal. For simple displays, there's Turtle Graphics, an easy-to-learn set of commands that produces surprisingly sophisticated patterns with a minimum number of program steps. Or, for better control and speed, you have all of QuickDraw's impressive power. QuickDraw is a toolset of programming, mostly in ROM, that Macintosh programs use to display windows, icons, shapes, and just about everything else you see on screen.

Unlike many computers, the Macintosh *draws* what it shows you. Boxes, lines, window borders, text, and other images appear as the result of commands that ultimately turn to white or black one or more of the display's 175,104 pixels, or *picture elements*.

This differs from conventional computers, which usually display text by encoding characters in memory. For example, to display the letter Q somewhere on the screen, a program would store the ASCII value 81 at a specific location. Having done that, the computer's video circuits read that and other character values to display pages of text.

The Macintosh is different. Instead of storing ASCII values in memory to display text, it graphically draws letters and symbols the same way it forms boxes, circles, and other shapes. Those shapes also have corresponding values in memory just as in a conventional display but, in this case, each of a pattern's memory bits represents a single display pixel.

There are several consequences of this approach to displaying visual information. For one, the Macintosh requires a large amount of memory to hold just one screen-full of graphics. On a traditional computer terminal where each memory byte can store one character, it takes only 2,000 bytes to hold all the ASCII values for an entire 80-column by 25-line screen. In contrast, because the typical Macintosh display has 512 horizontal by 342 vertical pixels, and because it takes one memory bit to hold each teensy dot, the Mac's display takes more than ten times

that amount of memory—to be exact, 21,888 bytes (512×342 divided by 8 bits per byte). To show something on this *bit-mapped* display, programs change individual bits in a screen *buffer*, a memory area that corresponds with the images you see.

The obvious disadvantage with a bit-mapped display is a loss of speed. Because the Macintosh has to read more than ten times the number of bytes per screen than most other computers, it potentially takes longer to redraw, or *refresh*, a page of text or graphics than it does on a traditional text-only terminal. Even worse, to keep the CRT (Cathode Ray Tube) phosphors from fading, as they quickly do unless constantly regenerated, the Macintosh must redraw its screen 60 times a second, a time-consuming juggling act from which it can never rest. (For those who care about accuracy, the exact rate is 60.15 hertz, or cycles per second.) Obviously, the amount of time it takes to scan 21,888 bytes of memory 60 times a second is a much greater slice of pie than reading only 2,000 bytes.

The extra effort is worth the trouble. On a text-only computer terminal, the hard-wired characters indelibly burned into the computer's circuits are all you get. You might be able to fake graphics with characters that look like lines and symbols, but you cannot change their bit patterns or combine them with other images. Of course, the Macintosh lets you do all of that and more. Because it draws everything you see, it paints italics, bold, and underlined text the same way it draws landscapes and animates figures. To the Macintosh, everything is a picture.

The one-to-one relationship between memory bits and display pixels is important to understand. But don't make the common mistake of thinking that bits and pixels are one and the same. They are two very different items. A memory bit is an electrical charge in a circuit somewhere in the RAM belly of your Mac. A pixel is a point of light, something you see on screen. The computer reads memory bits to know which points to turn on.

The difference might seem trivial on computers such as the Macintosh where pixels directly correspond with bits in memory. But it becomes even more important on color displays where many bits specify the color of individual pixels, no longer a one-to-one relationship. For example, the Macintosh II's video display stores color values in single *bytes*, using up to eight bits per visible pixel. By understanding now that bits and pixels are not necessarily equivalent, you'll find programming such displays easier in the future.

Another peculiarity you should know about is the way the black and white Macintosh interprets 0 bits as white pixels and 1 bits as black. This is opposite from usual graphics displays where 1 bits stand for white pixels. Many people see the Macintosh as behaving backwards in this department although, logically speaking, the Macintosh and traditional displays are the same—both interpret a 1 bit as significant, the value that produces something to see. Remember that it's what you *see* on the Macintosh that's different—usually black images on white backgrounds. As stored in memory, the images are no different than on most graphics computer displays.

TURTLE GRAPHICS

Seymour Papert and MIT coworkers invented Turtle Graphics in the late 1970s as a way to teach coordinate geometry to young people. By placing a turtle-like robot on a large sheet of paper, attaching a pen to the turtle's tail, and whizzing the little creature off in one direction or another, Papert discovered that complex patterns were easy to draw with only simple commands. Basically, there are four command types in the Turtle Graphics set:

PenDown—Put the pen down to start drawing

PenUp—Pull up the pen to stop drawing

Turn—Turn by a certain amount

Move—Move a certain distance

Variations on these commands turn the turtle to specific angles or move it to X,Y coordinates. Still other variations interrogate the turtle to get its current location and heading. To draw a box with such commands, you simply put the pen down and execute the following statements using a Pascal **FOR** loop:

```
PenDown;  
FOR side := 1 TO 4 DO  
BEGIN  
  Move( 50 );  
  Turn( 90 )  
END;
```

The most interesting fact about algorithms like this is that they work anywhere on the floor—or on the computer screen. You don't need to initialize variables or specify X,Y coordinates or line endings. You simply place the turtle at any location and tell it to move and turn until it draws a box.

Even more important, Turtle Graphics helps you to visualize what you want to draw. Instead of pondering formulas and mathematical equations, you envision the processes that make images. After all, what could be simpler than imagining a turtle scooting around leaving trails behind?

Of course, Turbo Pascal's turtle is a phantom. You never actually see the creature; you see only its trails. Some other languages that have Turtle Graphics, most notably Logo, do show a turtle figure, but not Turbo.

A STAR IS BORN

Turtle Graphics commands come packed inside a unit—a collection of precompiled routines and other definitions. There are commands to send the turtle to

specific locations, change its heading, clear the display, and so on. To use the commands, you tell Turbo to add the unit to your program.

Listing 3.1 demonstrates how to use the Turtle unit. It draws a five-point star in the center of a window, the same dumb-terminal from Chapter 2. Although this is a very simple example containing only 47 lines, it explains several features you will include in most of your own Turtle Graphics programs.

Type in the listing, save as STAR.PAS, and compile to memory. When you run it, you see a star in the center of the screen. Click the mouse button to end the program and return to Turbo.

Listing 3.1. STAR.PAS

```

1: { $O Programs:Turtle.F: }      { Send compiled code to here }
2:
3:
4: PROGRAM Star;
5:
6: (*
7:
8:  * PURPOSE : Draw a five-point star
9:  * SYSTEM  : Macintosh / Turbo Pascal
10: * AUTHOR  : Tom Swan
11:
12: *)
13:
14:
15:   USES
16:
17:       MemTypes, QuickDraw, OSIntf, ToolIntf, Turtle;
18:
19:
20:   CONST
21:
22:       Distance = 160;
23:
24:
25:   VAR
26:
27:       side : INTEGER;
28:
29:
30: BEGIN
31:   Home;
32:
33:   PenUp;
34:   SetHeading( 216 );
35:   Forwd( Distance DIV 2 );      { Center star in window }
36:
37:   SetHeading( 18 );
38:   PenDown;
39:   FOR side := 1 TO 5 DO        { Draw star }
40:     BEGIN
41:       Forwd( Distance );
42:       TurnRight( 144 );
43:     END; { for }
44:
45:   WHILE NOT Button DO { wait } { Wait for mouse click }
46:
47: END.
```

Star Play-by-Play

Line 1 directs the compiler to send its compiled code to the disk volume and folder Programs: Turtle.F:. Create this folder with the Finder, or change line 1 if you want to store the code file somewhere else. Whatever names you specify in this output compiler directive, Turbo uses them only when you compile to disk. It ignores line 1 when you compile directly to memory.

The **USES** clause in lines 15–17 tells Pascal to add the interfaces from five units, MemTypes, QuickDraw, OSIntf, ToolIntf, and Turtle. Turtle (the name of the Turtle Graphics unit) always comes after the preceding four names. You can use other units in Turtle Graphics programs—PasPrinter, for example, if you want to print text—but Turtle always follows the other four as shown here. The reason for this is that the Turtle unit itself uses the definitions in the other units. For example, to draw a line, Turtle routines such as **Forwd** call line drawing procedures in QuickDraw. And QuickDraw in turn uses the definitions in MemTypes, and so on.

Having told Pascal to include various units, you can then use their commands, constants, data types, and variables. For example, line 31 *homes* the turtle, sending it to the center of the window at coordinate (0,0) and turning it to face up (angle=0). If you take out the Turtle unit from the **USES** clause in line 17, the program would no longer compile because **Home** is not a command Pascal normally recognizes. It exists only in the Turtle unit.

To see the effect of **Home**, remove it and rerun the program. What happens? What does the result tell you? Removing commands this way is an excellent method to discover for yourself the effects they have. If you don't fully understand a sequence, take it out and observe what happens. Often, this simple debugging technique solves more mysteries than any other. But instead of erasing commands, and then having to retype them or reload the original file from disk, you can “comment them out,” meaning you turn them into a comment. For example, to comment out line 31, change it to this:

```
( * Home ; * )
```

I point this out to show you another small trick I use often. When I comment out a section of code, I always use the comment symbols (* and *). For normal comments (except for the program header) I use the alternate braces { and }. For examples, see lines 35, 39, and 45. Each of these lines ends, by my convention of using braces, in a regular comment. To comment such lines out, you could write (* in line 32 and *) in 36. Because they use two different kinds of brackets, the comments do not conflict with each other, an example of nested comments—one comment inside the other. This would not work if you used the same symbols, either (* and *) or { and } in both cases.

Lines 33–35 pull up the turtle's tail, set its heading to 216 degrees, and move it forward half the **Distance** constant value of 160, centering the star in the window. Lines 37–38 initialize the turtle's heading and put its pen down so that, when

it moves, it draws a line. Then the **FOR** loop (39–43) moves the turtle forward by a certain distance and turns it by 144 degrees (42) each time through the loop, drawing a star.

Notice that, except for the number of loops and the angle, the algorithm for drawing stars is no different from drawing boxes. It takes more steps (five instead of four) but it takes no additional statements to draw these two very different objects. This is typical of Turtle Graphics programs—a minimum number of program steps plus a curious ability to draw very different patterns using similar commands.

The final statement in Listing 3.1 uses the Toolbox Event Manager function **Button** to wait for you to click the mouse before the program ends. You activated the **Button** function, plus a few dozen other routines, by including the ToolIntf (Toolbox Interface) unit in the **USES** clause at line 17. You could put a **Readln** statement here in place of **Button** as you did in the textbook examples in Chapter 2, but then you'd see a flashing cursor on display along with the star. Using **Button** avoids displaying the cursor.

THE TWIRLING TURTLE

The next example, TWIRL.PAS in Listing 3.2, shows off Turtle Graphic's speed and takes advantage of the observation that a small number of similar steps can produce a variety of patterns. Type in the program the same way you typed in STAR.PAS. Compile to memory or to a disk code file.

When you run Twirl, read the brief introduction and click the mouse to begin. To stop the display and return to the Turbo editor, press the Return key (or any other key). Your only other job is to watch.

Listing 3.2. TWIRL.PAS

```

1: {$O Programs:Turtle.F: }           { Send compiled code to here }
2:
3:
4: PROGRAM Twirl;
5:
6: (*
7:
8:  * PURPOSE : Animated TurtleGraphics display
9:  * SYSTEM  : Macintosh / Turbo Pascal
10: * AUTHOR  : Tom Swan
11:
12: *)
13:
14:
15:     USES
16:
17:         MemTypes, QuickDraw, OSIntf, ToolIntf, Turtle;
18:
19:
20:     VAR
21:

```

```

22:      maxSides : INTEGER;
23:
24:
25:
26: PROCEDURE DrawPoly( len : INTEGER );
27:
28: { Draw a polygon at the pen position with sides = len }
29:
30:   VAR
31:
32:     side : INTEGER;
33:
34:   BEGIN
35:     PenDown;
36:     FOR side := 1 TO MaxSides DO
37:       BEGIN
38:         Forwd( len );
39:         TurnRight( 360 DIV MaxSides )
40:       END; { for }
41:     PenUp
42:   END; { DrawPoly }
43:
44:
45: PROCEDURE Graphics;
46:
47: { Display patterns by calling Box with various parameters }
48:
49:   VAR
50:
51:     steps, distance : INTEGER;
52:
53:   BEGIN
54:     Clear;
55:     steps := 1 + ( ABS( Random ) MOD 100 );
56:     maxSides := 1 + ( ABS( Random ) MOD 12 );
57:     FOR distance := 10 TO 1000 DIV maxSides DO
58:       BEGIN
59:         DrawPoly( distance );
60:         TurnRight( steps )
61:       END { for }
62:     END; { Graphics }
63:
64:
65: BEGIN
66:
67:   Writeln( 'Twirl' );
68:   Writeln( '-----' );
69:   Writeln;
70:
71:   Writeln( 'This program displays a variety of graphics patterns all' );
72:   Writeln( 'by rotating polygons around the screen center and ' );
73:   Writeln( 'increasing the size and orientation of each shape until' );
74:   Writeln( 'the screen is full.' );
75:   Writeln;
76:   Writeln( 'Press the Return key (or any other key) to stop the show.' );
77:   Writeln;
78:   Write( 'Click the mouse button to begin...' );
79:
80:   WHILE NOT Button DO { nothing };
81:
82:   HideCursor;
83:
84:   WHILE NOT Keypressed DO Graphics
85:
86: END.

```

Twirl Play-by-Play

Procedure **DrawPoly** (26–42) uses the same algorithm to draw a many-sided polygon as **Star** uses to draw stars. A **FOR** loop (36–40) repeats two statements, **Forwd** and **TurnRight**, the number of times global variable **maxSides** specifies. This moves the turtle forward according to the value of **len**, the procedure's only parameter. Notice that in line 39 the angle passed to **TurnRight** is (**360 DIV maxSides**), roughly ensuring that the final line ends back at the starting place, closing the shape. Of course, this works perfectly only for steps that divide evenly into 360. And, as you can see by reading the procedure, I use the word polygon loosely. If **len** equals 1, **DrawPoly** obviously does not draw a many-sided object. Also, values like 7 and 19 don't produce closed shapes. But, as you see when you run **Twirl**, there's no need to be so exacting in our definition of a polygon.

Notice also that **DrawPoly** puts the pen down when the procedure begins (35) and then pulls it up just before ending (41). This is a good plan to follow when designing your own graphics tools. If all routines leave the pen up, then you don't have to worry at other times whether moving the turtle will draw a line. You know it won't because all your tools cooperate, following guidelines you devise. On the negative side, the two **PenUp** and **PenDown** commands might make programs run more slowly, especially if they call **DrawPoly** many times.

The next procedure, **Graphics** (45–62), repeatedly calls **DrawPoly**, supplying various parameters to change polygon sizes and shapes. By now, you've probably run the program and seen the dramatic effect this has. The patterns twirl out at you starting at the center of the window and expanding until the display is full. Curves strangely appear even though the turtle draws only straight lines. And remember, everything you see is drawn by the **Forwd(len)** statement at line 38. What an amazing effect with such simple commands!

Line 54 shows how to clear the graphics window. Replace this line with the dumb terminal **ClearScreen** command and see what happens. The reason this doesn't work is that **ClearScreen** erases only text lines from the window. Always use **Clear** to erase the entire Turtle Graphics window out to the borders.

To randomly change the patterns, lines 55 and 56 set variables **steps** and **maxSides** to values selected at random by the QuickDraw toolset **Random** function, which returns an unpredictable value from the range –32,767 to +32,767. To keep **steps** and **maxSides** within a particular range, the program uses the typical approach of adding a minimum value (1 in line 56) to the value of **Random** modulo 12 (meaning the remainder after dividing by 12). But don't use this common formula:

```
n := Low + ( Random MOD High );
```

On many computers, that sets **n** to an unpredictable value ranging from **Low** to **High**. But, because QuickDraw's **Random** returns negative as well as positive values, if **Low** = 1 and **High** = 100 in the above formula, **n** could be any value from –98 to 100, not at all what you might expect. To get positive values only, pass the

value of **Random** to function **ABS** (absolute value) as in lines 55 and 56. This forces **Random** to return only positive numbers from 0 to 32,767.

Similar to the way **STAR.PAS** ends, line 80 waits for you to click the mouse button. Unlike **STAR.PAS**, where clicking the mouse ends the program, this starts Twirl's action but also causes the arrow cursor to appear. Therefore, line 82 hides the cursor with a call to the QuickDraw **HideCursor** procedure. To redisplay the arrow, call **ShowCursor**. If you do that, be aware that **HideCursor** counts the number of times you call it. Before the cursor reappears, you must call **ShowCursor** an equal number of times.

Just to be different, line 84 uses Turbo's **Keypress** function to end Twirl by pressing a key. If you would prefer clicking the mouse to end, replace line 84 with this:

```
WHILE NOT Button DO Graphics;
```

What happens when you do that? The answer is nothing, at least unless you're the fastest clicker in the land. When you click the mouse to start the program, line 84 executes before you can possibly release the mouse button and the program ends without displaying anything. One solution to this predicament is to wait for a release of the mouse button before starting the graphics display. To do that, insert this statement at line 81:

```
WHILE Button DO { wait for release };
```

Unfortunately, when you try this, you discover yet another problem. (Terrible how problems like these propagate, isn't it?) Now, when you click the mouse the first time, the program runs but, when you click it again, it doesn't end. (If you can't figure out how to end the program, hold the mouse button down between graphics frames—that should return you to Turbo.)

This demonstrates that programs don't see mouse clicks in the same way they see keypresses, an important distinction to remember. Keypresses go into a type-ahead memory area, or buffer, so that programs can check at any time whether you typed something earlier. Mouse clicks don't go into buffers. Instead, you have to sense them in real time. If the program doesn't check for a mouse click at the time that someone holds down the button, you can click all day and it won't have any effect. How can we solve this problem?

The solution is to check for mouse clicks inside the most time consuming loop in the program, the **FOR** loop at lines 57–61. If the program senses a mouse click there, it can call Turbo's **Exit** command to end the current procedure, jumping out of the **FOR** loop and letting the modified line 84 end when it senses the mouse button down. To make this change, insert the following **IF** statement between lines 58 and 59:

```
IF Button THEN Exit;
```


Solving problems by backtracking this way and revising previously correct procedures goes by the technical term *stepwise refinement*. In other words, after getting the program steps working the way you want, you may find it necessary to refine those steps to handle unforeseen situations that later arise.

QUICKDRAW GRAPHICS

The previous examples use Turbo Pascal's textbook interface, displaying everything in the dumb terminal window, which automatically appears when you compile and run programs. But you can also write graphics programs that draw directly on the Macintosh screen. Not only does this do away with Turbo's fixed window, it opens all QuickDraw routines and features to you and to your programs.

You pay for this newfound ability with added complexity. To fully use QuickDraw and other Macintosh toolsets requires you to follow certain rules and regulations. No longer can you write a simple program to draw lines by moving the turtle around. Instead, you have to initialize QuickDraw and tell it where you want it to draw (usually, but not always, the display). Then you can start drawing.

To make these added steps easier, it helps to have a shell—a do-nothing program that outlines the common steps most programs require. When you start a new program, you begin with a copy of the shell to which you add your own procedures, functions, and other declarations. The next section develops such a shell and then lists several examples that use it. To understand how it works, though, you first need to learn about the Macintosh coordinate system and a few QuickDraw data types.

Above the Coordinate Plane

If you've programmed other graphics computers, you'll find the Macintosh way a little different. The visible display is a mere chip off the entire coordinate block (more correctly called a *plane*) in which drawing occurs. You might think of the display as a sort of window (not the Macintosh kind of window) that sits above the plane and through which you view small sections of the entire surface. The full plane is 65,535 points wide and deep for a total of 4,294,836,225 ($65,535 \times 65,535$) points, every one available to you and your graphics programs.

Although large, the entire coordinate grid is not just sitting there in memory waiting for you to use one section or another. The plane is only a *logical* area into which you can draw shapes and other objects. If the entire plane were physically in memory, it would occupy over 500 *million* bytes—somewhat larger than the typical Macintosh holds.

You locate each coordinate point on this logical grid with two integers in the range $-32,767$ to $+32,767$. (Astute programmers will realize there is a missing negative value, $-32,768$, in this 16-bit integer range. Even though this is a legal

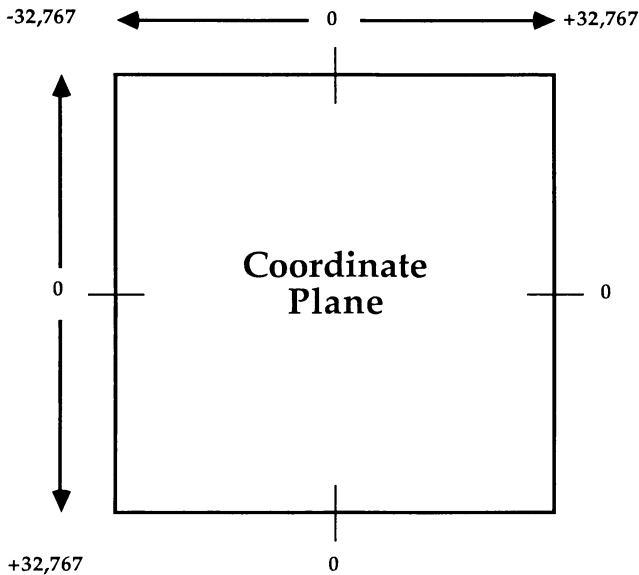


Figure 3.1 QuickDraw's coordinate plane is 65,535 pixels square with values ranging from $-32,767$ to $+32,767$ in both the vertical and horizontal axes.

integer, it's outside of QuickDraw's coordinate range.) As Figure 3.1 shows, negative values are to the top and left while positive values are to the bottom and right. Coordinate (0,0) is at dead center. On the horizontal axis, values follow the usual mathematics convention of putting negative X values to the left of zero and positives to the right. But on the vertical axis, values are opposite to mathematical convention, with negative values above zero and positive values below. (Figure 3.2 lists a Pascal function that converts Macintosh vertical coordinate values to standard mathematics convention.)

Because the Macintosh coordinate plane doesn't follow standard X,Y notation, it's best to label horizontal and vertical axes H and V rather than X and Y.

```
FUNCTION InvertV( v : INTEGER ) : INTEGER;

{ Invert vertical coordinate component v to convert Macintosh
  coordinates to standard mathematical convention }

BEGIN
  InvertV := 65536 - v
END; { InvertV }
```

Figure 3.2 Use this function to convert vertical coordinate values to standard mathematics convention where positive values are above zero and negative ones below—exactly the opposite in QuickDraw graphics.

Whenever examples in this book use H and V, or variations like H1 and vMax, you know they refer to points on the Macintosh coordinate plane.

Each coordinate point (H,V) is infinitely small—not a play on words, and not a reference to the fact that Macintosh pixels are tiny, as some people mistakenly assume. Points on the coordinate grid do not coincide directly with pixels on the screen. If you keep that simple fact in mind, you'll understand more than many people do about QuickDraw graphics.

Figure 3.3 shows why this is important. The figure represents a 4×5 section of the Macintosh coordinate grid containing four black pixels (the shaded cells). Notice that the coordinate values along the horizontal and vertical axes refer to the grid *lines*—not to the pixel columns and rows as they do on many other computer graphics screens. The leftmost shaded pixel is below and to the right of coordinate point (1,2) as marked by the arrow in the figure. The bottom shaded pixel has its top left corner at coordinate point (2,3).

This organization makes certain operations more logical than if coordinate values marked columns and rows instead of grid axes. For example, imagine a rec-

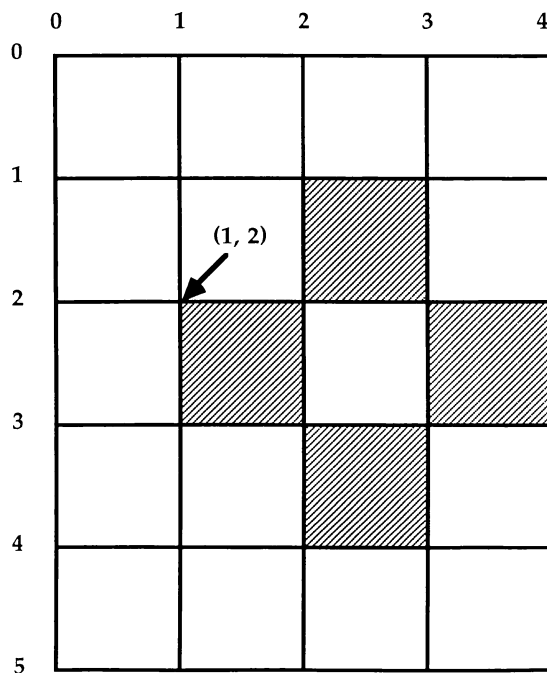


Figure 3.3 QuickDraw coordinates rest on the divisions *between* pixels, not on the rows and columns as in conventional computer graphics. As the arrow indicates, a pixel's coordinate locates its top left corner.

tangle around the four shaded pixels in Figure 3.3. How wide is it? How tall? Of course, the answer is three pixels. Now, if you write down the coordinate points of that rectangle's four corners, you see an interesting fact.

- (1,1)—Top left
- (4,1)—Top right
- (1,4)—Bottom left
- (4,4)—Bottom right

Notice that subtracting the left and right coordinate values ($4 - 1$) gives 3, the width of the rectangle in pixels. Likewise, subtracting the top from the bottom value also gives 3. This observation leads to a rule, one of the most important in Macintosh graphics programming: *Subtracting two coordinate points tells you how many pixels lie between those points.*

As a result, you don't have to use formulas such as $1 + (\text{right} - \text{left})$ to calculate object widths, as many computers require. To find the length of a line, you merely subtract its endpoint coordinate values, avoiding a common confusion that *Inside Macintosh* aptly calls "endpoint paranoia." An interesting side effect of this coordinate system is that you can specify zero-width objects. If two points have the same coordinates, there are no pixels between them. On a conventional system, there is one visible pixel at two points having the same coordinates even though subtracting the two vertical or horizontal coordinate values gives zero—a confusing contradiction the Macintosh neatly avoids.

Points and Rectangles

Two QuickDraw data types, **Point** and **Rect**, specify points and rectangular areas on the Macintosh coordinate plane. Figure 3.4 shows how QuickDraw defines these two record types.

VHSelect is an enumerated type with two components, **V** and **H**, representing vertical and horizontal axes values. **Point** is a variant record with two possible configurations, an example of a *free union*. The integers 0 and 1 (directly under **RECORD**) tell Pascal to treat the record fields either as two integers with the labels **v** and **h** (CASE 0) or as an array of integers indexed by type **VHSelect** (CASE 1). (If you read *Inside Macintosh*, you'll see that **VHSelect**'s components **V** and **H** are in lowercase. I use uppercase for these enumerated elements to distinguish them from the lowercase **v** and **h** integer fields in the **Point** record type. This follows the general rule in this book that variables start with lowercase letters while types and constants start with capitals.)

Point's free union structure specifies coordinate points two ways: either as separate **h** and **v** values or as an array of two integers indexed by **VHSelect** identifiers **H** and **V**. If you have a variable **AnyPoint** of type **Point**, you can use the following statements to assign coordinate (100,50) to it:

```

TYPE

  VHSelect = ( V, H );

  Point =
    RECORD CASE INTEGER OF

      0 : ( v   : INTEGER;
           h   : INTEGER );

      1 : ( vh  : ARRAY[ VHSelect ] OF INTEGER )

    END; { Point }

  Rect =
    RECORD CASE INTEGER OF

      0 : ( top    : INTEGER;
           left    : INTEGER;
           bottom  : INTEGER;
           right   : INTEGER );

      1 : ( topLeft : Point;
           botRight : Point )

    END; { Rect }

```

Figure 3.4 QuickDraw defines **Point** and **Rect** data types as free-union, variant records, allowing many ways to describe points and rectangles on screen.

```

AnyPoint.h := 100;
AnyPoint.v := 50;

```

Or, you could do the same thing with the **vh** array like this:

```

AnyPoint.vh[ H ] := 100;
AnyPoint.vh[ V ] := 50;

```

It may seem useless to have two different ways to specify points but there's a good reason for the **vh** array even though two array index operations appear excessive merely to assign two integers. Usually, you'll use the prior method and assign values to the **v** and **h** integer fields. But with the **vh** array, you can use a variable as the index and let the program logic decide whether to affect a horizontal or a vertical component. For example, you might have the procedure in Figure 3.5a.

The first procedure (Figure 3.5a) adds **amount** to the **vh** array field, using parameter **select** as an index and, therefore, selecting either the horizontal or the vertical component—without the procedure itself knowing in advance to which component it adds **amount**. The second procedure (Figure 3.5b) adds amount to the **v** or **h** integer fields but, in this case, an **IF** statement decides which addition to make.

Although these examples are not necessarily ideal ways to add values to coor-

```

PROCEDURE MovePoint1( VAR p : Point;
  select : VHSelect; amount : INTEGER )

BEGIN
  p.vh[ select ] := p.vh[ select ] + amount
END; { MovePoint }

```

(a)

```

PROCEDURE MovePoint2( VAR p : Point;
  select : VHSelect; amount : INTEGER );

BEGIN
  IF select = V
    THEN p.v := p.v + amount
    ELSE p.h := p.h + amount
  END; { MovePoint }

```

(b)

Figure 3.5 Although the two procedures do the same job, **MovePoint1** (a) uses array indexing to assign coordinate values to point **p** rather than record field designations as in **MovePoint 2** (b).

ordinates, they demonstrate what many programmers often forget: one or two array index operations like `p.vh[select]` in some circumstances may be more efficient than two or four record field specifications like `p.v` and `p.h`.

Another variant record, **Rect** (see Figure 3.4), defines a rectangular area on screen. Rectangles figure in many QuickDraw operations. You'll use them to draw various shapes—not only rectangular ones—and to define regions, which specify arbitrarily shaped portions of the visible display. You'll use them also to define the size and location of Macintosh windows.

Rectangles are never visible on their own. Because their borders lie on infinitely thin axes grid lines, they merely define areas on the coordinate plane. You can fill rectangles with patterns, outline them, and use them to draw circles and arcs. The rectangles define only where and how large such shapes are to be—they have no visible pattern themselves.

The **Rect** data type is a free union, similar in that way to **Point**. As Figure 3.4 shows, there are two ways to assign and use its fields. The first way (CASE 0) offers four integer variables: **top**, **left**, **bottom**, and **right**, marking the coordinate points of the top-left and bottom-right rectangle corners. Alternatively, you can assign **Point** records to fields **topLeft** and **botRight** (CASE 1). This is useful when you already have two coordinate points and you want to use them to define a rectangle, perhaps as the result of someone clicking the mouse at two screen locations. A third way to assign rectangle fields is to call the QuickDraw procedure **SetRect**. As an example of these methods, the following statements define a rectangle **r** encompassing the entire grid in Figure 3.3.

```

r.top := 0; r.left := 0;
r.bottom := 5; r.right := 4;

```

You could do the same thing by assigning values to **Point** fields **topLeft** and **botRight** this way:

```
r.topLeft.v := 0; r.topLeft.h := 0;
r.botRight.v := 5; r.botRight.h := 4;
```

As you can see, such assignments are confusingly complex. For extra clarity, call procedure **SetRect** as follows.

```
SetRect( r, 0, 0, 4, 5 ); { Left, Top, Right, Bottom }
```

Although this looks neater, you might think it to be less efficient than the other direct assignments to rectangle fields. In fact, there is so little difference between the methods, the best plan is to use **SetRect**—it makes your programs more readable.

Always remember that **Rect** variables can enclose exactly one pixel. For example, a **Rect** with **topLeft** = (1,2) and **botRight** = (2,3) encloses the single leftmost shaded pixel in Figure 3.3. Similarly, you can have empty rectangles that enclose no bits. A **Rect** value with **topLeft** = (3,5) and **botLeft** = (3,5) is a legitimate construction but has no width or height. These are important concepts to keep in mind.

By the way, some people have trouble remembering **SetRect**'s parameter order: Left, Top, Right, and Bottom. This is confusing especially because **Rect**'s fields are in the more natural sequence: Top, Left, Bottom, and Right. Having mixed up these parameters too many times, I finally remembered the correct order with the help of a mental trick. It may seem silly but there's a famous brewery town in our state where they make a favorite beer, Rolling Rock. The town's name happens to be Latrobe (LTRB, get it?) and that's the way I remember **SetRect**'s parameter order. I can't imagine why I chose this particular mnemonic late one night. I must have been thirsty.

A GRAPHICS SHELL

All QuickDraw routines draw in something called a **GrafPort**—a complex record that keeps track of various parameters affecting what you see on screen—and often a lot of things you don't see. Of the **GrafPort**'s 25 fields, you'll rarely need to use more than one or two. For most operations, you'll call QuickDraw routines that properly assign and use **GrafPort** field values. You won't assign those values yourself. (The complete definition for **GrafPort** is in the *Guide* and *Inside Macintosh*.)

Listing 3.3 is a shell that you can use for most QuickDraw graphics programs. It properly initializes a **GrafPort**, erases the screen to a black background, and draws a white border around the outer edges. Although this is contrary to the usual Macintosh black on white display, it's a popular format—especially for games and other graphics programs. Type in the listing and save as GRAPH SHELL.PAS. As in the other examples, change line 1 to compile to different volume and folder names. When you run the program, it waits for you to click the mouse button—but it doesn't do anything except outline the screen.

GraphShell Play-by-Play

Line 2 turns off Turbo's standard library units, automatically used by textbook style programs. Because of this, GraphShell specifies four toolbox units, **Memtypes**, **QuickDraw**, **OSIntf**, **ToolIntf** (17). Usually, these four are the minimum you need to write QuickDraw graphics programs. **Memtypes** defines a few standard data types but contains no code or calls to ROM routines as do most units. **QuickDraw**, of course, defines the Macintosh drawing procedures, functions, data types, and variables. **OSIntf** is the operating system interface. It handles memory management, I/O operations, device drivers, and other low-level jobs. Although it contains many definitions you'll rarely if ever use, you still must include it. Contrasting **OSIntf** is the fourth unit, **ToolIntf**, which contains routines that you'll probably use more often than many others. Although this chapter uses only a few **ToolIntf** commands, later examples rely heavily on this toolset to display and manipulate windows.

Line 22 defines a single variable **gPort** of type **GrafPort**. Because this creates the port as a global variable in the program, it permanently takes up space above the stack—as do all global variables, which are located in memory specifically set aside for this purpose. Another way to create a **GrafPort** is to put it on the heap, the memory area where programs create *dynamic structures*, meaning those that it creates when the program runs as opposed to those that you create when you write the program.

Figure 3.6 lists a function, **MakeGrafPort**, that you can use to create new **GrafPorts** on the heap. Notice that it returns a pointer to a **GrafPort** (of type **GrafPtr**)—not a **GrafPort** record. This pointer holds the address of the **GrafPort** record that the function creates on the memory heap.

One danger of using **MakeGrafPort** is that it might fragment the heap, a condition that leaves holes in memory between other objects. Those holes can cause programs to run out of memory if they need to create memory areas bigger than the largest available hole. This is an especially critical condition on the Macintosh where desk accessories and toolbox routines compete with your program for memory. Later, we'll see how to prevent this problem.

To modify Listing 3.3 to use the **MakeGrafPort**, insert the function into GraphShell at line 23 and change three other lines to read as follows:

```
22 gPort : GrafPtr;
34 gPort := MakeGrafPort;
35 r := gPort^.portRect;
```

In programs that reference **gPort** as a variable, add a caret (^) as shown in modified line 35. Doing this is called “dereferencing the pointer,” meaning that you tell Pascal to use not the address stored in pointer **gPort**, but the object to which it points—in this case, a **GrafPort** record on the heap. If you're unclear about doing this, read a Pascal tutorial's chapter on using pointers.

```

FUNCTION MakeGrafPort : GrafPtr;

{ Create a GrafPort variable on the heap and return its address
  as the function result. }

VAR

    tempPtr : GrafPtr;

BEGIN
    tempPtr := GrafPtr( NewPtr( SIZEOF( GrafPort ) ) );
    OpenPort( tempPtr );
    MakeGrafPort := tempPtr
END; { MakeGrafPort }

```

Figure 3.6 Use this function to create **GrafPorts** on the heap rather than as variables on the stack.

If you decide to create a global variable **GrafPort** as in Listing 3.3, the only disadvantage is a limit of about 32K for all program globals combined. Because **GrafPorts** take up only 108 bytes, though, a few of them permanently in memory aren't likely to cause severe shortages.

SetupScreen (25–42)

Continuing with GraphShell (Listing 3.3), procedure **SetupScreen** initializes the graphics display, erases it to a black background, and draws a white border. **SetupScreen** calls seven QuickDraw procedures—ones you'll undoubtedly use frequently in your own programs.

Even though you created the **GrafPort** variable (either as a global variable or on the heap), you need to initialize it. **OpenPort** (34) does this by setting the **GrafPort** fields to default values and by creating a few other internal structures that QuickDraw uses to keep track of the clipping and visible regions (**clipRgn** and **visRgn** fields in **GrafPort**). You never need to access these items directly, but you should be aware of their existence. The clipping region defines a boundary around which no drawing appears. If you tell QuickDraw to draw outside of this area, it automatically *clips* the lines (and other shapes) at the boundaries. The visible region is used mostly by windows to keep track of visible and hidden portions behind other windows on top. Because these structures are regions and not rectangles, they can have any size and shape within the confines of the Macintosh coordinate system. It is difficult and even unwise to manipulate regions in Pascal programs and, for that reason, you should let QuickDraw handle them on its own.

Line 36 of GraphShell calls **PenPat** to set the drawing pattern for subsequent QuickDraw commands. Patterns are small, eight-byte objects, which Pascal defines as follows.

```

TYPE
    Pattern = PACKED ARRAY[ 0 .. 7 ] OF 0 .. 255;

```

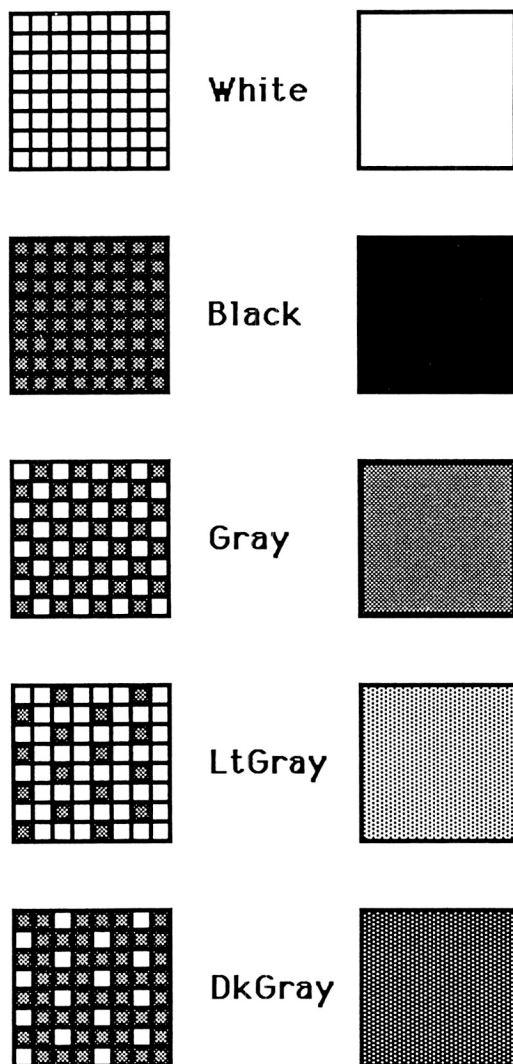


Figure 3.7 QuickDraw's five standard 8×8 pen patterns on the left paint the shades on the right.

This makes a 64-bit object which QuickDraw uses for all drawing operations. Figure 3.7 illustrates five of these, defined as global variables in the QuickDraw unit. The boxes on the left are blow-ups of the pixel arrangements that produce the standard shades on the right. You can also create your own patterns by setting the bytes in a **Pattern** variable any way you like and then passing that variable to **PenPat** or to any other QuickDraw routine that takes a **Pattern** as a parameter.

After setting **PenPat** to Black, **PaintRect** (37) erases the display, filling it with

black bits. Next, another call to **PenPat** (38) returns to the default pattern, **White**. Calling **FrameRect** (39) then draws a white border around the display area.

To preserve this border, lines 40–41 reduce rectangle **r** by one pixel in from each border (reducing also both the width and height of the display by two). First line 40 calls **InsetRect**, which subtracts one from its left and right borders, and one from the top and bottom. You can use **InsetRect** also to expand rectangles. For example,

```
InsetRect(r, -5, -10);
```

increases rectangle **r**'s width by 10 and its height by 20 pixels. After changing a rectangle, you can then restrict drawing to within its borders by passing it to procedure **ClipRect**, as line 41 does. By doing this, **GraphShell** protects the white outline from overdrawing by other commands.

DoGraphics (45–55)

As you can see in the listing, **DoGraphics** simply waits for you to press the mouse button. It does this only so you can see what **GraphShell** does—admittedly not much on its own. Line 52 shows where to insert **QuickDraw** routines to display graphics. After describing the rest of the shell, I'll explain in more detail how to do that.

GrafShell Body (58–65)

Before using the **QuickDraw** toolset, you must initialize it. Line 59 does this by calling **InitGraf**, passing the address of **thePort**, a **GrafPtr** pointer to a **GrafPort** record. This **GrafPort** tells **QuickDraw** routines how and where to draw shapes, lines, patterns, text and other patterns. It defines the characteristics of visible graphics as well as the lower-level details of where in memory drawing takes place. **GrafPorts** limit **QuickDraw**'s view into memory the way a ship's porthole limits a sailor's view onto the sea. As we proceed, I'll explain many of the details that make up **GrafPort** records. But for now, all you need to know is that graphics programs need one before they can do any drawing.

Line 34 in procedure **SetupScreen** opens **gPort** with a call to procedure **OpenPort**, which initializes the **GrafPort**'s fields. This has no visible effect on the display but serves only to prepare the port for subsequent drawing commands. Notice the at-sign (@) in front of **gPort** in line 34. In Turbo Pascal, this means “pointer to,” indicating that the statement **OpenPort(@gPort)** passes not the contents of the **gPort** record but a pointer to its address in memory. You probably know that in Pascal the caret (^) also means “pointer to.” But carets are strictly for defining pointer data types as in:

```
TYPE  
  IntPointer = ^INTEGER;
```

and for *dereferencing* pointers to variables on the heap as in:

```
VAR
  iPtr : IntPointer;
BEGIN
  iPtr^ := 100
END.
```

To distinguish between these types of pointers and a variable's address, Turbo Pascal uses an at-sign. This is typical in Macintosh programming. Many procedures such as **OpenPort**, **GetPort**, and **SetPort** require pointers to **GrafPort** variables in this way.

But, you might wonder, what is **thePort** (59)? It's not a variable in GrafShell but rather a variable in QuickDraw, which defines **thePort** (**TYPE GrafPtr**) for every program that uses the unit. This pointer locates the current **GrafPort**. Every QuickDraw routine that doesn't take a **GrafPtr** parameter gets the information it needs from **thePort**'s fields. Lines, for example, always appear in **thePort**. Changing text fonts affects text in **thePort**.

You change the current port by passing a different **GrafPtr** to QuickDraw's **SetPort** routine. You also change it when you call **OpenPort** as in line 34, which sets **thePort** to point to the shell's **gPort** variable (22) and initializes its fields. This completes the initialization, sets up the port, and readies QuickDraw.

Line 60 initializes the Font Manager, required only if you are going to display text. (It does no harm, however, to initialize it anyway.)

The next two lines (61–62) initialize the mouse pointer cursor and then immediately hide it from view. Even though the program hides the cursor, it still must initialize it as in the listing. The reason for this is if someone manages to start a program while the cursor is a shape other than the usual arrow—as sometimes happens to super-fast clickers—when the program ends, it will redisplay the cursor as that shape instead of an arrow. This may last only a few seconds but, if you want an arrow to appear later, always initialize the cursor by calling **InitCursor**.

Line 63 calls **FlushEvents** to remove any waiting keypresses or mouse clicks from an internal waiting area called the *event queue*. In this queue, which operates as a list where items go in one end and come out the other like cars queuing up at the gas pump, the Macintosh operating system inserts records that describe events such as keypresses and mouse clicks. Programs receive and respond to these events to update displays and perform other actions. Chapter 4 describes how to write event-driven programs that cooperate with the operating system to move windows, use desk accessories, and choose commands from pull-down menus. Here, we'll use the event queue in a simpler fashion—perfectly allowable in pure graphics programs.

Finally, lines 64–65 call the GraphShell's **SetupScreen** procedure and then **DoGraphics**, displaying graphics with commands you insert between **REPEAT** and **UNTIL** (50–54).

Saving Graphics in MacPaint Files

To save your masterpiece graphics displays in MacPaint disk files, you need to modify GraphShell (Listing 3.3). After making these modifications, pressing Command-Shift-3 creates a MacPaint picture file of the screen. To make the changes, insert the following **DoGraphics** routine at lines 45–55.

```
PROCEDURE DoGraphics;
VAR
  theEvent : EventRecord;
BEGIN
  REPEAT
    { insert QuickDraw commands here }
  UNTIL GetNextEvent( mDownMask+keyDownMask, theEvent )
END; { DoGraphics }
```

This new procedure works by calling **GetNextEvent**. The first parameter specifically checks for mouse button and key presses in the event queue. The second parameter is the event record. (Chapter 4 explains what events are and how to use them in programs.) This allows the operating system to recognize Command-Shift-3 as a command to create a MacPaint file containing the screen contents and still let you end graphics programs by clicking the mouse.

PENS AND LINES

For most operations, QuickDraw uses an imaginary pen that touches the intersection of one point on the coordinate plane. You can move this pen to new locations, tell QuickDraw to draw shapes there, and set various parameters that change the way the pen works.

Two routines examine and change the **PenState**, a data type that describes the current pen. Use **GetPenState** to examine the pen's settings and **SetPenState** to change them. Figure 3.8 shows the four fields in a **PenState** record, which these routines take as a parameter. Field **pnLoc**, a **Point** record, is the location of the

```
TYPE
  PenState =
    RECORD
      pnLoc   : Point;      { Pen coordinate (h,v) }
      pnSize  : Point;      { Pen width (h) and height (v) }
      pnMode  : INTEGER;    { Bit transfer (display) mode }
      pnPat   : Pattern     { Drawing pattern to use }
    END; { PenState }
```

Figure 3.8 The drawing pen has four distinguishing characteristics, stored in a QuickDraw **PenState** record with this structure.

pen on the coordinate plane. In GraphShell, the pen starts at location (0,0), usually the top left corner of the visible screen. The **pnSize** field describes the width and height of the pen in pixels. The **h** component of **pnSize** equals the width; the **v** component equals the **height**. (Don't mistake **h** [horizontal] for the pen height!)

When changing pen parameters, you have two choices. You can use routines in Table 3.1 to modify various settings, or you can store parameters in a **PenState** record and pass them to **SetPenState**. Probably, it's best to use routines like **PenMode** and **PenSize** rather than storing directly into **PenState** fields. Most of the time, you'll use **GetPenState** and **SetPenState** to save and restore a pen's configuration as in this fragment:

```
VAR
  pnState : PenState;

BEGIN
  GetPenState( pnState );

  { ... routines that change
    the current pen state... }

  SetPenState( pnState )
END;
```

Drawing lines and dots is easy with **Line**, **LineTo**, **Move**, and **MoveTo** routines (see Table 3.1). A few examples help clarify how they work. Use a copy of GraphShell (Listing 3.3) and insert the new procedure from Figure 3.9 in place of **DoGraphics** (45–55). When you run the program, you see two lines crossing from corner to corner of the display.

Table 3.1 QuickDraw's line and pen tools.

```
PROCEDURE GetPen( VAR pt : Point );

PROCEDURE GetPenState( VAR pnState : PenState );

PROCEDURE Line( dh, dv : INTEGER );

PROCEDURE LineTo( h, v : INTEGER );

PROCEDURE Move( dh, dv : INTEGER );

PROCEDURE MoveTo( h, v : INTEGER );

PROCEDURE PenMode( mode : INTEGER );

PROCEDURE PenNormal;

PROCEDURE PenPat( pat : Pattern );

PROCEDURE PenSize( width, height : INTEGER );

PROCEDURE SetPenState( pnState : PenState );
```

```

PROCEDURE DoGraphics;

BEGIN

    WITH screenBits.bounds DO
    BEGIN
        LineTo( right, bottom );
        MoveTo( right, top );
        LineTo( left, bottom )
    END; { with }

    REPEAT
    UNTIL Button
    END; { DoGraphics }

```

Figure 3.9 Replace procedure **DoGraphics** in Listing 3.3 with this programming to draw a simple test pattern.

The **WITH** statement in this example uses QuickDraw's global **screenBits** variable **bounds** rectangle to obtain the Macintosh screen boundary coordinates. Whenever you need to determine the limits of the Macintosh screen, get them from **screenBits.bounds**. Never assume that the screen is so many pixels wide or tall. Otherwise, your programs won't work on different size screens likely to show up in future models of the computer.

It takes three statements in Figure 3.9 to draw the two lines from corner to corner. **LineTo** draws a line in the current pen pattern (white in this case) starting from the pen's position and extending in a straight line to the bottom right corner of the display. The pen automatically moves to this new location after drawing the line. Similarly, **MoveTo** moves the pen but doesn't draw anything. The program uses this technique to move the pen to new starting places, here the top right corner. A second **LineTo** then finishes the drawing.

There's no restriction on the pen size. You can make it fat or skinny. It can be a block or a rectangle of any practical dimension. To change the pen size, insert this statement above the **WITH** statement in the new **DoGraphics** procedure:

```
PenSize( 5, 5 );
```

Run the new program and the lines are much heavier now that the pen is five pixels wide and tall. Try other values. Change the width and height and observe the effect. Try (25,25) to see a problem. The lines are no longer centered in the corners. What makes them go askew?

Seeing this problem demonstrates something you should remember about the pen. QuickDraw always draws as though the pen touched a rectangle's upper left corner. The size of that rectangle is the size of the pen. As Figure 3.10 illustrates, visible drawing effects (shown by the darkened squares) occur to the right and down from the pen's (h,v) coordinate. Therefore, drawing with large pen sizes doesn't center the lines at the pen point. Use the procedure in Figure 3.11 to fix the problem and

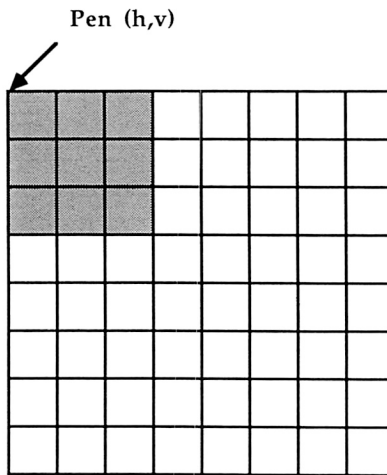


Figure 3.10 The pen's ink flows down and to the right of its coordinate (h,v).

center lines by positioning them so they meet the corners dead center. Be sure you understand how this works. Experiment with the values until you do. And always remember where the pen is in relation to the lines it draws.

Besides moving the pen, you can use **GetPen** to find out its current location (see Table 3.1). **GetPen** returns a **Point** record equal to the (h,v) coordinate of the pen. You can also use other patterns by passing them to **PenPat**. Try adding one of the following statements to **DoGraphics**:

```
PenPat ( gray );
PenPat ( ltGray );
PenPat ( dkGray );
```

To see the effect, it helps to have a rather fat pen, maybe 15 or 20 pixels wide. Even more interesting is to make up your own pen patterns. As you learned earlier, a pattern is simply an array of eight byte values in the range 0 to 255 (Figure 3.7). The values represent the bits that the pen ink draws in. Add a pen pattern **p**, an integer variable **i**, plus these statements to **DoGraphics** before the **WITH** statement:

```
VAR
  i : INTEGER;
  p : Pattern;

FOR i := 1 TO 6 DO
  p[i] := 145;
p[0] := 255;
p[7] := 255;
PenPat ( p );
```

In this and future fragments, I list only the necessary statements in order to save space and avoid too many duplications. I assume that you know the **VAR**

```

PROCEDURE DoGraphics;

{ Fill in with your own graphics routines }

CONST

    width = 16;

BEGIN

    PenSize( width, width );

    WITH screenBits.bounds DO
    BEGIN
        LineTo( right, bottom - ( width DIV 2 ) );
        MoveTo( right - width, top );
        LineTo( left, bottom - width )
    END; { with }

    REPEAT
    UNTIL Button
    END; { DoGraphics }

```

Figure 3.11 Change the pen size to draw fat or skinny lines. Replace procedure **DoGraphics** in Listing 3.3 with this programming for an example.

declaration goes in the procedure's **VAR** section. (You don't need to duplicate the keyword **VAR** if the procedure already has one—I use it here merely for reference.) I assume also that you know to add the statements to the body of the program. If you have no trouble with this fragment, you'll have no trouble with others.

Figure 3.12 shows why you see crosshatches after designing your own Pattern

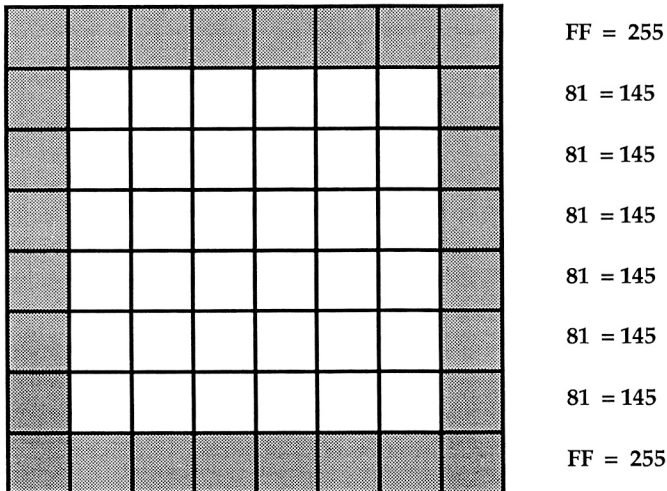


Figure 3.12 To design your own pen patterns, fill in squares in an 8 × 8 grid and calculate the values each row represents in hex, converting that number to decimal (the far right column here).

variable **p**. Each row of the pen pattern corresponds to a binary value, which in turn corresponds to hex and integer equivalents as shown at the right of the Figure. To draw in a custom pattern, simply insert these values into a **Pattern** array, pass it to **PenPat** and start drawing.

DRAWING TEXT

One of QuickDraw's great features is its ability to draw text in as many different fonts and styles as you can install in your System file. Just pick whatever font you want and you can display characters in that style anywhere on screen.

Table 3.2 lists the major text procedures in QuickDraw. Use **TextFont** to select a new font according to the partial list of constants in Table 3.3. If you try to use a font that doesn't exist, nothing bad happens—QuickDraw simply ignores your request.

To display text, insert the procedure in Figure 3.13 in a copy of GraphShell, replacing procedure **DoGraphics**. This new procedure begins by setting the text mode to **notPatCopy**. The text mode is the method by which QuickDraw copies bits to the display. (For more information about this subject, see the heading Drawing

Table 3.2 QuickDraw's text tools.

FUNCTION	CharWidth(ch : CHAR)	:	INTEGER;
PROCEDURE	DrawChar(ch : CHAR)	:	
PROCEDURE	DrawString(s : Str255)	:	
FUNCTION	StringWidth(s : Str255)	:	INTEGER;
PROCEDURE	TextFace(face : Style)	:	
PROCEDURE	TextFont(font : INTEGER)	:	
PROCEDURE	TextMode(mode : INTEGER)	:	
PROCEDURE	TextSize(size : INTEGER)	:	

Table 3.3 QuickDraw's font constants.

CONST			
systemFont	= 0;	toronto	= 9;
applFont	= 1;	cairo	= 11;
newYork	= 2;	losAngeles	= 12;
geneva	= 3;	times	= 20;
monaco	= 4;	helvetica	= 21;
venice	= 5;	courier	= 22;
london	= 6;	symbol	= 23;
athens	= 7;	taliesin	= 24;
sanFran	= 8;		

```

PROCEDURE DoGraphics;

  BEGIN
    TextMode( notPatCopy );
    MoveTo( 50, 75 );

    DrawString( 'Text and the single character' );

    REPEAT
      UNTIL Button
    END; { DoGraphics }

```

Figure 3.13 QuickDraw draws *everything*, even text as this **DoGraphics** replacement procedure for Listing 3.3 demonstrates.

Modes a little later in this chapter.) Here, the program changes the text mode to force QuickDraw to display white characters on the graphics screen's black background. You could display black on white characters by erasing the display to white in procedure **SetupScreen** (change line 36 from Black to White). In that case, you do not have to change text modes to display characters.

Try changing to different fonts by adding **TextFont** statements with one of the constants from Table 3.3. Use **systemFont** to display characters in the style that you normally see for menu bars, commands, and window titles. Usually, the **systemFont** is Chicago. Use **applFont** to display text in the style most programs use for text inside windows, usually Geneva. Use Monaco for monospaced text that Turbo and most other program editors display.

You can also change *point size*, a phrase that refers to a typesetter's unit measure for character height in 1/72-inch increments. Because pixels on the Macintosh are very nearly as tall as a single point, text on display closely matches printed text in the same point size. To display 18-point Helvetica, for example, you could write:

```

TextFont( helvetica );
TextSize( 18 );

```

In addition to fonts and sizes, you can select among various styles by passing a **Style** set to procedure **TextFace**. QuickDraw defines **TYPE Style** as in Figure 3.14.

You can use any combination (or none) of the **StyleItem** elements such as **bold**

```

TYPE

  StyleItem = ( bold, italic, underline, outline, shadow,
               condense, extend );

  Style = Set of StyleItem;

```

Figure 3.14 Text can have any of the seven **StyleItems** shown here plus one—plain. Represent combinations of styles as **Style** sets, for example, [bold, outline].

and **outline** inside set brackets to change text styles. For example, add the next three statements to **DoGraphics** (Figure 3.13) between **MoveTo** and **DrawString**:

```
TextFont( systemFont );
TextFace( [ outline, shadow, underline ] );
TextSize( 18 );
```

One thing to remember about point sizes in QuickDraw is that some values have corresponding bit patterns in the System file and others do not. For instance, there might not be a 24-point **NewYork** font although there may be a 12-point size. In that case, QuickDraw scales the nearest font size up or down in an attempt to meet your request for a certain text size. Usually, the results are blocky and often unreadable. Also, any point size smaller than 9 is probably too tiny to show clearly.

Two routines draw characters in the font, size, and style you choose. Figure 3.13 shows how to display entire strings or string variables with the **DrawString** procedure. Similar to **DrawString** is **DrawChar**, which draws a single character.

Because most fonts are proportional, different letters have different widths. As a general rule, a capital M is probably the widest character and a lowercase l the thinnest, but not always. Combined with a variety of fonts, sizes, and styles, it's often necessary to know just how much space a character or string occupies on the graphics display.

To find out, use functions **CharWidth** and **StringWidth** to calculate the pixel-width of characters and strings. You can use this information to avoid running text off the edge of the display—a harmless although poor-looking condition. (Remember that the coordinate plane is 65,535 points square. You can attempt to draw outside of the visible display with no bad effect; you just won't see the result.)

Listing 3.4 uses **CharWidth** and other QuickDraw routines to display a font's character set. Type it in and save as CHARS.PAS. When you run it, you'll see a display of all the characters available in the font that line 98 chooses. (The square boxes indicate characters that have no corresponding symbol.)

Listing 3.4. CHARS.PAS

```
1: { $O Programs:Graphics.F: }      { Send compiled code to here }
2: { $U- }                          { Turn off standard library units }
3:
4:
5: PROGRAM Chars;
6:
7: (*
8:
9:  * PURPOSE : Display a font's character set
10:  * SYSTEM  : Macintosh / Turbo Pascal
11:  * AUTHOR   : Tom Swan
12:
13:  *)
14:
15:
16:   USES
17:
18:       Memtypes, QuickDraw, OSIntf, ToolIntf;
```

```

19:
20:
21:     VAR
22:
23:         gPort : GrafPort;
24:
25:
26: PROCEDURE SetupScreen;
27:
28: { Initialize display for upcoming graphics }
29:
30:     BEGIN
31:         OpenPort( @gPort );           { Open new graphics port }
32:         PenPat( Black );               { Select drawing color }
33:         PaintRect( gPort.portRect )   { Fill screen with black }
34:     END; { SetupScreen }
35:
36:
37: FUNCTION TextHeight : INTEGER;
38:
39: { Returns the height in pixels }
40:
41:     VAR
42:
43:         fInfo : FontInfo;   { Holds information about current font }
44:
45:     BEGIN
46:         GetFontInfo( fInfo );
47:         WITH fInfo DO
48:             TextHeight := ascent + descent + leading
49:         END; { TextHeight }
50:
51:
52: FUNCTION EndOfLine( ch : CHAR ) : BOOLEAN;
53:
54: { Returns TRUE if drawing this character would move pen beyond
55: the right screen border }
56:
57:     VAR
58:
59:         penLocation : Point;
60:
61:     BEGIN
62:         GetPen( penLocation );
63:         WITH penLocation DO
64:             EndOfLine :=
65:                 ( h + CharWidth( ch ) > screenBits.bounds.right )
66:         END; { EndOfLine }
67:
68:
69: PROCEDURE CrLf;
70:
71: { Simulate a carriage return, line feed for the current font }
72:
73:     VAR
74:
75:         penLocation : Point;
76:
77:     BEGIN
78:         GetPen( penLocation );
79:         MoveTo( 0, penLocation.v + TextHeight )
80:     END; { CrLf }
81:
82:
83: PROCEDURE DoGraphics;
84:
85: { Display ASCII a font's characters }

```

(continued)

```

86:
87:   CONST
88:
89:       PointSize = 24;
90:
91:   VAR
92:
93:       ch : CHAR;
94:       penLocation : Point;
95:
96:   BEGIN
97:       TextMode( notPatCopy );
98:       TextFont( systemFont );
99:       TextSize( PointSize );
100:
101:       MoveTo( 0, pointSize );
102:
103:       FOR ch := chr(0) TO chr(255) DO
104:           BEGIN
105:               IF EndOfLine( ch )
106:                   THEN CrLf;
107:               DrawChar( ch )
108:           END; { for }
109:
110:           REPEAT
111:               UNTIL Button
112:           END; { DoGraphics }
113:
114:
115:   BEGIN
116:       InitGraf( @thePort );           { Initialize Quickdraw }
117:       InitCursor;                     { Make sure cursor level = 0 }
118:       HideCursor;                     { Make cursor invisible }
119:       FlushEvents( everyEvent, 0 ); { Erase any pending events }
120:       SetupScreen;                    { Initialize display }
121:       DoGraphics                      { Draw whatever you want }
122:   END.

```

Chars Play-by-Play

You'll recognize some of the Listing from earlier examples. Lines 1–23 and 115–122 are identical to GraphShell (Listing 3.3). **SetupScreen** (26–34) is similar but doesn't draw a white border around the display. Notice that here there is no need to copy the **GrafPort portRect** field to a temporary variable in order to reduce it by one pixel, protecting the border as in GraphShell. Instead, line 33 just paints the port's enclosing rectangle to erase the screen to all black.

TextHeight, EndOfLine, and CrLf (37–80)

Two functions and one procedure are three tools you might want to extract for your own graphics programs that display text. Function **TextHeight** (37–49) returns the total height in pixels of the current font's point size. Because characters like y and j extend below the base line of capital letters, a font's display height is not equal to its point size. To calculate the true height, line 46 calls **GetFontInfo**, which returns a **FontInfo** record with the structure in Figure 3.15. Adding the as-

TYPE

```

FontInfo =
RECORD
    ascent    : INTEGER;    { Pixels above base line (at pen) }
    descent   : INTEGER;    { Pixels below base line (at pen) }
    widMax    : INTEGER;    { Maximum width of any character }
    leading   : INTEGER     { Pixels between single-spaced text lines }
END; { FontInfo }

```

Figure 3.15 A **FontInfo** record describes the font's size in four integer fields.

cent (the height in pixels above the pen position, or *base line*), **descent** (the height in pixels below the base line), and **leading** (the number of pixels between single-spaced text lines) for this font gives its actual height in pixels.

Function **EndOfLine** (52–66) returns **TRUE** if drawing character **ch** would move the pen beyond the right border. Use this tool to test the pen position before drawing characters. If it returns **TRUE**, move the pen to the next line (or do something else) to prevent chopping characters in half.

The function works by examining the pen's location, calling **GetPen** (62) for its current coordinate. The horizontal value of this coordinate (**h**) plus the character width indicates whether there is enough room to the right of the pen. To make this calculation, **EndOfLine** calls **CharWidth** (65) and uses the global **screenBits.bounds** rectangle to locate the screen's right edge.

To start a new display line on conventional terminals, you simply type **Writeln** which sends carriage return and line feed control characters to the video terminal. The Macintosh doesn't work that way. Because there are no fixed character positions and because fonts can be any size and have proportional-width symbols, programs have to simulate carriage returns and line feeds in software.

Procedure **CrLf** (69–80) shows one way to do this. It first examines the pen's current position, calling **GetPen** to load a local **Point** record variable, **penLocation**. After that, it moves the pen by calling **MoveTo**, passing 0 as the new horizontal coordinate value and the pen's vertical component (**v**) plus the height in pixels of the current font (**TextHeight**).

DoGraphics (83–112)

DoGraphics displays a font's character set. It first selects the **notPatCopy** text mode (97) to display white characters on the graphic screen's black background. Next, it selects the font (98). Try other font names in place of **systemFont** (see Table 3.3). Line 99 sets the text size to the constant **PointSize**, which you can change by using a different value at line 89. The default value 24 looks good even for fonts that don't have corresponding bit images for this size, forcing QuickDraw to scale the image from the nearest size actually stored on disk. But you can use any point size you want.

Line 101 moves the pen to the first display position, with **h** equal to zero and **v** equal to the font's point size. This moves the pen down from the top of the display the number of pixels of the tallest character above the font's base line. Usually,

the point size equals this value—the **ascent** field in the font info record as explained for function **TextHeight** (37-49). To make the program more correctly locate the first base line, read the **FontInfo** record and use the **ascent** field in place of constant **pointSize** in line 101.

The **FOR** loop (103-108) draws each character, starting with ASCII value 0 and ending at 255. On conventional text-only terminals, visible ASCII characters normally range from 32 to 126 or 127. Other values either go unused or represent controls that cause display hardware to perform certain actions. On the Macintosh, all ASCII characters are visible although some may not have bit patterns representing characters. There are no control characters in QuickDraw text—drawing a carriage return (ASCII 13) displays a symbol; it doesn't move the cursor to the start of a line.

Line 105 checks whether drawing each successive character **ch** will fit between the current pen position and the screen's right border. If not, **CrLf** (106) positions the pen one line down at far left. The next line (107) draws the single character by calling QuickDraw's **DrawChar** routine.

USING RECTANGLES

Rectangle records (type **Rect**) have a variety of uses in QuickDraw. You've seen how they define rectangular areas on the screen, but there is much more you can do with them. You can use them to draw boxes, either filled or unfilled; to draw ovals and circles within their borders; and even to draw wedges that you might need in a pie chart program.

Table 3.4 lists the QuickDraw procedures that either change rectangles or use them in calculations. We've already seen **SetRect**, which simply assigns left, top, right, and bottom values to a **Rect** record.

OffsetRect and **InsetRect** change the shape of a rectangle after you assign its

Table 3.4 QuickDraw's rectangle tools.

```
PROCEDURE EraseRect ( r : Rect );

PROCEDURE FillRect ( r : Rect; pat : Pattern );

PROCEDURE FrameRect ( r : Rect );

PROCEDURE InsetRect ( VAR r : Rect; dh, dv : INTEGER );

PROCEDURE InvertRect ( r : Rect );

PROCEDURE OffsetRect ( VAR r : Rect; dh, dv : INTEGER );

PROCEDURE PaintRect ( r : Rect );

PROCEDURE SetRect ( VAR r : Rect; left, top, right, bottom : INTEGER );
```

corner coordinates. **OffsetRect** repositions the rectangle according to the two parameters, **dh** and **dv**. For example, this moves a rectangle **r** right 25 pixels:

```
OffsetRect ( r, 25, 0 );
```

And this moves **r** up 75 pixels:

```
OffsetRect ( r, 0, -75 );
```

Positive **dh** values move right; positive **dv** values move down. Negative **dh** values move left; negative **dv** values move up. This is logical because **OffsetRect** adds **dh** to **r**'s left and right fields and **dv** to its bottom and top.

InsetRect changes a rectangle shape in a different way. In this case, QuickDraw adds **dh** to the top value and subtracts it from the bottom, moving the top and bottom coordinates in toward the center. Similarly, it adds **dv** to the left and subtracts it from the right, moving those borders toward the center, too. The following statement shrinks a rectangle **r** by 20 pixels vertically and 50 pixels horizontally.

```
InsetRect ( r, 25, 10 );
```

Notice that the values you pass to **InsetRect** are one half the total number of pixels by which you want to shrink the rectangle in either the horizontal or vertical direction. This is because two borders move by the amounts **dh** and **dv**.

To expand rectangles, use negative values. This statement expands **r** 64 pixels horizontally, while not changing its height:

```
InsetRect ( r, -32, 0 );
```

After setting a rectangle or modifying its values, you can use it in drawing commands. For example, **FrameRect** draws a line connecting a rectangle's four corners. You already used **FrameRect** and **PaintRect** to outline and paint the screen to all black in GraphShell (Listing 3.3, lines 37, 39). Another way to clear the screen follows:

```
BackPat ( Black );  
EraseRect ( gPort.portRect );
```

If you pass a **Pattern** variable to **BackPat**, **EraseRect** uses it to fill a **Rect** area. This example passes the **GrafPort**'s enclosing rectangle, **portRect**, but you can erase any other rectangle, too. Because this method does not require changing the pen pattern, you can improve GraphShell by inserting these two statements in place of those at lines 36 and 37.

Pass a rectangle to **InvertRect** and QuickDraw reverses all pixels inside that area. Black pixels become white and vice versa. For a sample of what this routine does, insert the following statement at line 109 in Listing 3.4, CHARS.PAS.

```
InvertRect( gPort.portRect );
```

The difference between **FillRect** and **PaintRect** is that **FillRect** takes a pattern as a parameter; **PaintRect** uses the current pen pattern to fill rectangles. But there is a more important difference that's not obvious. **PaintRect** fills rectangles using the pen's current transfer mode, changing not only the pattern you see but the method by which QuickDraw copies bits to the display. (For details about transfer modes, see the section Drawing Modes later in this chapter.) **FillRect** always uses the **patCopy** mode, which simply copies patterns bit for bit into a rectangle without combining those bits in any way with other images already there.

DRAWING CURVED SHAPES

QuickDraw uses a novel idea to draw circles. Instead of specifying radii and center points, you simply declare a rectangle and then call a routine to draw an oval within its borders. This simplifies the usual way to draw curves on computer displays. It also lets you draw both circles and ovals with the same routine.

Table 3.5 lists five procedures that draw ovals inside rectangles. Notice the similarity of these names and those in Table 3.4. Because of this, their operations should be obvious. **EraseOval** erases a circular shape inside **Rect r** to the current background pattern (**BackPat**). **FillOval** fills an oval with a specific pattern. **FrameOval** outlines a circle inside its rectangle. **InvertOval** reverses all bits inside

Table 3.5 QuickDraw's oval tools.

PROCEDURE EraseOval(r : Rect);
PROCEDURE FillOval(r : Rect; pat : Pattern);
PROCEDURE FrameOval(r : Rect);
PROCEDURE InvertOval(r : Rect);
PROCEDURE PaintOval(r : Rect);

Table 3.6 QuickDraw's round rectangle tools.

PROCEDURE EraseRoundRect(r : Rect; ovalWidth, ovalHeight : INTEGER);
PROCEDURE FillRoundRect(r : Rect; ovalWidth, ovalHeight : INTEGER;
pat : Pattern);
PROCEDURE FrameRoundRect(r : Rect; ovalWidth, ovalHeight : INTEGER);
PROCEDURE InvertRoundRect(r : Rect; ovalWidth, ovalHeight : INTEGER);
PROCEDURE PaintRoundRect(r : Rect; ovalWidth, ovalHeight : INTEGER);

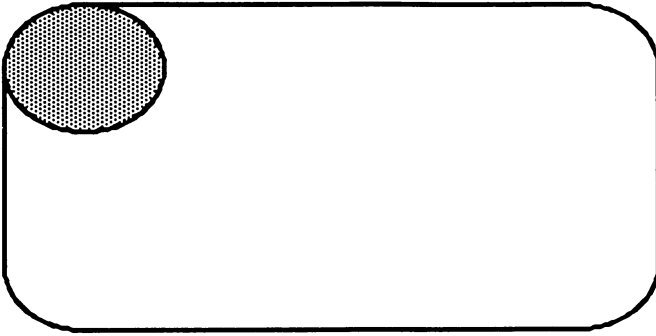


Figure 3.16 To alter the curvature in round rectangles, vary the width and height of an imaginary oval fitting snugly inside each corner.

```

PROCEDURE DoGraphics;

{ Illustrate relationship between ovals and round rectangles }

CONST

    ovalWidth = 75;
    ovalHeight = 60;

VAR

    box, oval : Rect;
    theEvent : EventRecord;

BEGIN

    SetRect( box, 80, 70, 430, 270 );
    FillRect( box, white );
    InsetRect( box, 25, 25 );

    oval := box;
    WITH box DO
    BEGIN
        oval.right := left + ovalWidth;
        oval.bottom := top + ovalHeight
    END; { with }

    PenPat( Black );
    PenSize( 2, 2 );
    FillOval( oval, ltGray );
    FrameOval( oval );
    FrameRoundRect( box, ovalWidth, ovalHeight );

    REPEAT
        SystemTask
    UNTIL GetNextEvent( mDownMask+keyDownMask, theEvent )
END; { DoGraphics }

```

Figure 3.17 Replace Listing 3.3's **DoGraphics** procedure with this routine to draw the design in Figure 3.16, demonstrating round rectangles.

the oval. And **PaintOval** is like **PaintRect**—it paints ovals in the current pen pattern and transfer mode.

Another kind of curved shape is a *round rectangle*, an oxymoron if ever there was one. To draw one, use the procedures in Table 3.6. Because their names are similar to those in Tables 3.4 and 3.5, you should have little trouble understanding what they do. (Try them in GraphShell if you are unsure.)

Figure 3.16 explains the relationship between round rectangle corners and the parameters **ovalWidth** and **ovalHeight** in each of the five **RoundRect** procedures. QuickDraw rounds the rectangle as though it contained a shaded oval nested inside its corners. (Although only one oval is shown here, the same curve applies to all four corners.) As you can see, changing the oval's shape affects the amount of curve in each corner and, therefore, changes the degree of rectangular roundness. Figure 3.17 lists the **DoGraphics** routine that produced the round rectangle illustration. It also shows how to use some of the QuickDraw commands described earlier. Add it to a copy of GraphShell, replacing the **DoGraphics** procedure there.

DRAWING MODES

QuickDraw understands eight methods of combining one group of bits with another. By changing modes, you affect the way QuickDraw displays bits on screen by altering how it combines them with pixels already there. The four basic methods are: Copy, Or, Xor (eXclusive or), and Bic (Bit clear). Inverting each of these—changing white bits to black and vice versa before drawing—gives a total of eight possible modes. Each mode is similar to standard Boolean logic operations. For that reason, they are best illustrated by the truth tables in Table 3.7.

In the truth tables, 0 stands for a white pixel and 1 stands for black, the same as their bit values in memory. Column S is the source bit, from a pen pattern for

Table 3.7 Drawing-mode truth tables.

Copy			Or		
S	D	R	S	D	R
0	0	0	0	0	0
1	0	1	1	0	1
0	1	0	0	1	1
1	1	1	1	1	1
Xor			Bic		
S	D	R	S	D	R
0	0	0	0	0	0
1	0	1	1	0	0
0	1	1	0	1	1
1	1	0	1	1	0

Table 3.8 QuickDraw's drawing-mode tools.

```

PROCEDURE PenMode( patMode : INTEGER );

PROCEDURE TextMode( srcMode : INTEGER );

```

example; D is the destination, probably the display; and R is the result—what you see on the display after combining two bits according to the rules for this table.

The Copy mode transfers source bits directly to the destination, ignoring whatever is already there. (Column R is an exact copy of S.) Mode Or displays a white bit only if both the source and destination are white. If either the source, destination, or both, are black, so is the result. Xor mode displays black if either the source or destination is black—but not both. In this mode, if both bits have the same value, they turn white in the drawing. Xor mode also has the interesting property of restoring the original destination when you redraw the same pattern twice. This is a useful technique in animations where patterns move overtop of others without disturbing them.

In the Bic (bit clear) mode's truth table, the result is always white unless the source is white and the destination black. Bic mode is useful in clearing areas (called “punching a hole”) on screen for displaying icons (see Chapter 7).

Two QuickDraw routines use transfer modes to affect the way other routines draw graphics and text. (See Table 3.8.) **PenMode** changes the way the pen pattern appears. After calling **PenMode**, the pen behaves according to one of the truth tables in Table 3.7. **TextMode** similarly affects the way text routines display characters.

Both procedures take a single **INTEGER** parameter that stands for one of the eight basic drawing modes. *Inside Macintosh* identifies both parameters as **mode**. To avoid confusing text and pen modes, here I use **patMode** (pattern mode) in **PenMode** and **srcMode** (source mode) for **TextMode**. (The word *source* refers to the way QuickDraw copies bits from a source area, in this case the bits that make up a character in a font.) This makes it easier to remember which of the two groups of constants in Table 3.9 apply to which procedure. Use the constants on the left with **PenMode**; the ones on the right with **TextMode**.

Table 3.9 QuickDraw's drawing mode constants.

```

CONST

{ Pattern modes                      Source modes          }
{ -----                          -----              }
patCopy    = 8;                      srcCopy     = 0;
patOr      = 9;                      srcOr       = 1;
patXor     = 10;                     srcXor      = 2;
patBic     = 11;                     srcBic      = 3;
notPatCopy = 12;                     notSrcCopy  = 4;
notPatOr   = 13;                     notSrcOr    = 5;
notPatXor  = 14;                     notSrcXor   = 6;
notPatBic  = 15;                     notSrcBic   = 7;

```

```

PROCEDURE DoGraphics;

{ Test text source transfer modes }

CONST

    message = 'TextMode Test String.';
    hText   = 10;
    vText   = 50;

BEGIN
    BackPat( ltGray );
    EraseRect( gPort.portRect );

    TextSize( 32 );
    TextMode( srcXor );

    MoveTo( hText, vText );
    DrawString( message );

    REPEAT UNTIL Button;

    MoveTo( hText, vText );
    DrawString( message );

    WHILE Button DO {wait for release};
    REPEAT
    UNTIL Button
    END; { DoGraphics }

```

Figure 3.18 Text transfer modes affect text appearance by changing the way QuickDraw combines font bit patterns with pixels already on display. Replace **DoGraphics** in Listing 3.3 with this procedure for a demonstration.

Experimenting with various modes is easy—just insert **TextMode** and **PenMode** commands into any drawing program and use an appropriate constant from Table 3.9. One of the most useful modes is Xor (eXclusive or). With this almost magical drawing mode, you can draw on top of patterns, remove the drawing, and automatically restore the original graphics beneath. An experiment demonstrates how this works. Type procedure **DoGraphics** from Figure 3.18 into a copy of GraphShell (Listing 3.3), replacing the procedure at lines 45–55.

When you run the program, you see a light gray background with a message near the top. Click the mouse and the message disappears. Click again to end the program. The procedure works by setting **TextMode** to **srcXor**. Then, it displays the test message, waits for you to click the mouse, and redisplay the same message to erase it and restore the background. This works no matter what graphics you draw over. With the exclusive or mode, redrawing any shape twice restores whatever was there before.

Try all eight source modes from Table 3.9 in **DoGraphics**, replacing **srcXor** in procedure **TextMode**. As you can see, many varieties of text styles, backgrounds, and effects are possible simply by changing the way QuickDraw combines character

bits with graphics already on display. You can achieve similar effects with line drawing by passing pattern modes (Table 3.9) to **PenMode**.

A third QuickDraw procedure, **CopyBits**, transfers bit patterns on a more fundamental level. With this procedure, you copy one area of memory to another, applying one of the eight transfer modes to the result. Usually, you put **CopyBits** to work copying figures from unseen memory areas to the video buffer. Doing that repeatedly, overlaying figure over figure, is an easy way to animate graphics. The process resembles the way a professional animator shoots successive frames and then projects them at high speed to make a cartoon film.

BIT MAPS

Before using **CopyBits**, you need to understand two more QuickDraw structures, bit maps and regions. A bit map is an area in memory that translates, or *maps*, those bits as though they were pixels on display. In other words, by specifying memory areas as bit maps, you are in effect saying, “If these bits were displayed, this is how I want QuickDraw to arrange them.”

The display itself is one giant bit map. In this case, the bits in memory and the pixels coincide—you actually see the bits in the arrangement you specify. But you can make bit maps anywhere in memory and then tell QuickDraw to draw there instead of on the visible screen. After drawing in your offscreen bit map, you tell **CopyBits** to copy the drawing to the display where you can see it. In many cases, this action produces a smoother effect than drawing directly to the visible screen.

Figure 3.19 defines QuickDraw’s **BitMap** record. Three fields hold a starting address in memory, the number of bytes in one row, and an enclosing rectangle, which imposes a coordinate system on the bit map. The starting address, expressed as a pointer (**Ptr**) to signed memory bytes with values -128 to $+127$, must be even because of the way the Macintosh’s 68000 processor (and descendants) use memory in two-byte chunks, or *words*, at a time. The **baseAddr** pointer tells where in memory the bit map starts. For similar reasons, **rowBytes** must also be an even number. It describes how many bytes are in a single row. The **bounds** rectangle imposes a

```

TYPE

    BitMap =
        RECORD
            baseAddr    : Ptr;           { Starting address in memory }
            rowByte     : INTEGER;       { Number of bytes in one row }
            bounds      : Rect           { Coordinate system }
        END;

```

Figure 3.19 A **BitMap** record tells QuickDraw where and how to draw in memory.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
2000																
2010																
2020																
2030																
2040																
2050																
2060																
2070																

Figure 3.20 Bit maps, as defined by **BitMap** records, organize memory into two-dimensional arrays, possibly wasting a few unneeded bits shown here as the shaded cells on the right.

coordinate system on the bit map, telling in effect which pixels appear in which positions relative to others.

Bit maps are easier to understand with an illustration. Figure 3.20 shows the relation between the following programming and bytes in memory.

```
VAR bits : BitMap;

WITH bits DO
BEGIN
  baseAddr := POINTER($2000);
  rowBytes := 2;
  SetRect( bounds, 0, 0, 13, 8 )
END; { with }
```

The first job is to assign an area of memory, here starting at hexadecimal address \$2000. In practice, you never assign actual addresses as in this illustration. Instead, you'll normally declare variables to hold bit maps and assign their addresses to the **baseAddr** field. Or, you can use the memory manager to allocate memory to hold bit maps, a technique we'll use later.

Field **rowBytes** equals the number of full, eight-bit bytes in one row of bits. As you can see from Figure 3.20, the example bit map has exactly two bytes per row (16 bits, 0 to F). This organization is completely up to you. A bit map can have any number of bytes per row, up to 32,766, as long as the value is even. Whatever value you choose, you must be certain that it contains the number of bits per row that you want to use.

The third field in a bit map record is a **bounds** rectangle. Its four fields, left, top, right, and bottom, establish three bit map characteristics.

- The number of bits in one row
- The number of bits in one column
- The bit map's size in bytes

Figure 3.20 outlines in bold the rectangle corresponding to the horizontal coordinate values left = 0 and right = 13. (The shaded cells are outside of this area.) You can use any values as long as the number of bits they encompass (right-left) is no greater than (rowBytes*8). Notice that the illustration wastes three bits per row (the shaded cells). As far as QuickDraw is concerned, these bits do not exist.

The **bounds** rectangle declares also the number of bits in one column, equal to the rectangle's height (bottom-top). In the example, the height is eight bits. Together, **bounds** and **rowBytes** exactly limit the amount of memory the bit map occupies according to the following formula for a BitMap **b**:

```
WITH b DO
  bytes := ( bottom - top ) * rowBytes;
```

Applying this formula to the illustration in Figure 3.20 gives 16 bytes. Notice that changing the width of the **bounds** rectangle has no effect on the size of the bit map in memory—it merely tells QuickDraw how many bits in each row of bytes to use. But changing the **bounds** height and the number of bytes per row does affect the bit map size. For that reason, after setting up a bit map, never change its parameters without careful thought. It's your responsibility to ensure that the parameters you specify actually correspond to a reserved area in memory large enough to hold the entire bit map.

REGIONS

Another important QuickDraw structure is a region. In much the same way that **Rect** variables define rectangular areas on the coordinate plane, regions define areas with no particular shape. A region can be rectangular, circular, pear shaped, or as convoluted as an island's shoreline. In fact, islands in the ocean resemble regions on the coordinate plane. Both enclose freeform areas in their worlds.

Regions have many uses in Macintosh software. Windows use them to update portions of themselves uncovered when you move other windows aside. You can use them to limit, or *clip*, drawing to irregular shaped areas on the screen. And you can also put a region to work as a sort of graphics collector into which you draw closed shapes with standard QuickDraw routines. After collecting your drawing into a region, QuickDraw can outline it, fill it, and perform other operations to affect how the region appears.

Listing 3.5 demonstrates one way to use regions. Insert the listing in place of GraphShell's **DoGraphics** (lines 45-55, Listing 3.3). Save as REGIONS.PAS. When you run it, type the space bar (or any other key) to change patterns. Hold down the mouse button while you type any key to end the program.

Listing 3.5. REGIONS.PAS

```

1: FUNCTION Keypressed : BOOLEAN;
2:
3: { TRUE if a key was pressed }
4:
5:   VAR
6:
7:       theEvent : EventRecord;
8:
9: BEGIN
10:   Keypressed := GetNextEvent( KeyDownMask, theEvent )
11: END; { Keypressed }
12:
13:
14: PROCEDURE Randomize;
15:
16: { Start new random sequence }
17:
18:   VAR
19:
20:       time : LONGINT;
21:
22: BEGIN
23:   GetDateTime( time );
24:   RandSeed := time
25: END; { Randomize }
26:
27:
28: FUNCTION randH : INTEGER;
29:
30: { Return a horizontal coordinate value at random }
31:
32: BEGIN
33:   WITH screenBits.bounds DO
34:     randH := ABS( Random ) MOD ( right - left )
35:   END; { randH }
36:
37:
38: FUNCTION randV : INTEGER;
39:
40: { Return a vertical coordinate value at random }
41:
42: BEGIN
43:   WITH screenBits.bounds DO
44:     randV := ABS( Random ) MOD ( bottom - top )
45:   END; { randV }
46:
47:
48: PROCEDURE DoGraphics;
49:
50: { Draw shapes by collecting in region.  Type space bar (or any key)
51:   to change picture.  Hold mouse down and type space bar to end. }
52:
53:   VAR
54:
55:       rh      : RgnHandle;   { Handle to region }
56:       tempRect : Rect;       { For drawing ovals }
57:       i       : INTEGER;     { Loop control variable }
58:
59: BEGIN
60:
61:   Randomize;           { Start new random sequence }
62:   BackPat( black );    { Background is black }
63:

```

```

64:      REPEAT
65:        EraseRect( gPort.portRect );      { Erase old image }
66:
67:        rh := NewRgn;                      { Start new region }
68:        OpenRgn;
69:
70:        FOR i := 1 TO 25 DO                { Invisibly draw things }
71:          BEGIN
72:            SetRect( tempRect, randH, randV, randH, randV );
73:            FrameOval( tempRect )
74:          END; { for }
75:
76:        CloseRgn( rh );                   { Stop collecting graphics }
77:        FillRgn( rh, ltGray );             { Display region }
78:        DisposeRgn( rh );                 { Remove from memory }
79:
80:        WHILE NOT Keypressed DO {wait}
81:
82:      UNTIL Button
83:
84:    END; { DoGraphics }

```

Regions Play-by-Play

Although the four miscellaneous tools at lines 1–45 have nothing to do with regions, it's convenient to introduce them here. Function **Keypressed** (1–11) returns **TRUE** after someone types a key. It calls **GetNextEvent** to request whether the operating system received any keyboard events. (Chapter 4 explains events in more detail.) Earlier you learned that Turbo has its own **Keypressed** routine. Unfortunately, it's available only to textbook programs that use Turbo's dumb terminal window. The function here adds the same ability to QuickDraw graphics and fully charged programs with windows and pull-down menus.

Function **Randomize** (14–25) starts new random sequences by reading the current time (23) and assigning it to global variable **RandSeed** (24). Because the Macintosh represents the time as the number of elapsed *seconds* from January 1, 1904, **Randomize** sets **RandSeed** to a different value every time it runs. To see why this is necessary, turn lines 23–24 into a comment and run the program. View a few pictures and end. Then rerun. Because it doesn't start a new random sequence, the program shows the same pictures again.

Some programmers attempt to start new random sequences by assigning **RandSeed** to **RandSeed** like this:

```
RandSeed := RandSeed;
```

This never works. Random sequences are predictable if you know the starting point. Because the starting random number is the same unless you change it, using **Random** to initialize **RandSeed** simply uses the second value in a predictable sequence. Assigning the time to **RandSeed**, though, does set it to an unpredictable value and always generates new random sequences.

Two functions, **randH** (28–35) and **randV** (38–45), return horizontal and vertical coordinate values at random, limited to the Macintosh display width and height. Use them to select coordinates at random. Because they use the global **screenBits** variable, they work with any size Macintosh display. For example, you could draw randomly placed lines with the following statements, which you can add to a copy of GraphShell along with **randH** and **randV** from Listing 3.5:

```
PenPat( white );
WHILE NOT Button DO
    LineTo( randH, randV );
```

DoGraphics (48–84)

Variable **rh** is a region handle—a special kind of pointer—which QuickDraw uses to locate in memory where it saves region information. (Remember, the exact nature of that information is unimportant. Your program simply needs to keep track of the handle for QuickDraw's use.) Lines 61–62 start a new random sequence and set the background to black for clearing old images with **EraseRect** (65) each time you press a key.

Lines 67–68 show the correct way to start a new region. **NewRgn** reserves memory for QuickDraw's use and returns a handle, which the program saves in variable **rh**. Line 68 opens the region, telling QuickDraw to begin saving lines and shapes as the program draws them. This also hides the pen, meaning that no visible drawing takes place while QuickDraw collects its regional data.

Because the pen is now invisible, the **FOR** loop (70–74) doesn't actually draw images. Instead, it defines the borders of the region for QuickDraw by calling **SetRect** and **FrameOval** to create a number of ovals of various sizes and shapes.

After that, line 76 closes the region, making the pen visible again and telling QuickDraw to stop collecting regional information. **FillRgn** (77) fills the region with a light gray pattern, showing the entire drawing at one time. Everything you see on screen happens when this single instruction executes. Line 78 calls **DisposeRgn** to remove the regional data from memory. Always dispose your region handles after you're done using them. Otherwise, the program risks running out of memory.

USING SCREENBITS

A most important bit map variable, **screenBits**, tells QuickDraw the location and size of the Macintosh display. Its **baseAddr** field points to the first byte of memory that holds the bit image of what you see on screen. If you need to locate this address, never assume that it starts at a fixed location. Different size Macintoshes place the display bit map at different locations. Always use **screenBits.baseAddr** to get the starting address of display memory.

ScreenBits' other two fields, **rowBytes** and **bounds**, define the size of the visi-

ble display, giving it a system of coordinates that match the width and height of the **bounds** rectangle. You'll rarely refer to **rowBytes**, but you'll use **bounds** often. Because future Macintosh models might have larger displays, never assume that the screen is 512 pixels wide by 342 pixels high, as it is in Macintosh and Macintosh Plus models. Instead, get the width and height from **screenBits** this way:

```
WITH screenBits.bounds DO
BEGIN
    width := right - left;
    height := bottom - top
END; { with }
```

Related to **screenBits** is a **GrafPort** field, **portBits**. When you create a **GrafPort** (see Listing 3.3, line 34), QuickDraw copies **screenBits** to the **portBits** field, making every new **GrafPort** initially use the entire display. But you are free to reassign this bit map to another area in memory, draw something there, and then copy the results to the visible screen.

When you create new **GrafPorts**, QuickDraw also copies the **screenBits.bounds** rectangle to the **GrafPort portRect** field. In other words, new **GrafPort** fields **portBits.bounds** and **portRect** are identical. Don't be confused by this apparent duplication. The **portRect** field can change size—for example, to limit drawing to a window's contents or another area on screen. But the **portBits** bit map **bounds** field never changes size—it defines the coordinate system physically imposed on memory. Changing **portBits.bounds** can have the disastrous effect of telling QuickDraw to draw anywhere in memory, even over your program and variables! Be sure you understand the difference between **portBits** and **portRect**. **PortBits** defines the physical in-memory bit map. **PortRect** defines coordinates relative to the coordinate plane. **PortBits** can never exceed the memory allocated to the bit map. **PortRect** can be as large or as small as you need.

An example helps explain how to use bit maps to draw images off screen and then copy those images to the display. Understanding the example will help you to understand the relationship between **PortBits** and **PortRect**. It animates a walking figure by copying successive frames from bit maps to the screen. Using a copy of GraphShell (Listing 3.3), replace procedure **DoGraphics** (45–55) with all of Listing 3.6 and save as ANIMATE.PAS.

Listing 3.6. ANIMATE.PAS

```
1: PROCEDURE DoGraphics;
2:
3: { Fill in with your own graphics routines }
4:
5:   CONST
6:
7:       maxFrameNumber = 5;      { Number of frames in animation }
8:       maxRows        = 22;     { Number of 2-byte rows in a frame }
9:
```

(continued)

```

10:  TYPE
11:
12:      Frame = ARRAY[ 1 .. 22 ] OF INTEGER;    ( Exactly 44 bytes )
13:
14:  VAR
15:
16:      theEvent      : EventRecord;
17:      frameNumber   : INTEGER;
18:      srcBits       : BitMap;
19:      destRect      : Rect;
20:      frames        : ARRAY[ 1 .. maxframeNumber ] OF Frame;
21:
22:
23:  PROCEDURE PrepareFrames;
24:
25:      { Create off-screen bit maps.  Each bit map is an individual
26:        frame in the animation. }
27:
28:  BEGIN
29:      StuffHex( @frames[1],
30:        Concat( '0000', '0030', '0078', '00FC', '03FF',
31:          '00F8', '00FC', '00F8', '00F0', '07C0',
32:          '0FE0', '1FE0', '33E0', '67E6', '67FE',
33:          '0FF8', '1F80', '3F80', '79C8', '70F8',
34:          '7070', '3820' ) );
35:
36:      StuffHex( @frames[2],
37:        Concat( '0000', '0000', '00C0', '01E0', '03F0',
38:          '0FFC', '03E0', '03F0', '03E0', '03C0',
39:          '0F80', '1FC0', '3FC0', '37C0', '67E8',
40:          '67F8', '07C0', '0F80', '7DC0', '78C0',
41:          '60C0', '60F0' ) );
42:
43:      StuffHex( @frames[3],
44:        Concat( '0000', '0060', '00F0', '01F8', '07FE',
45:          '01F0', '01F8', '01F0', '00E0', '01E0',
46:          '03E0', '07C0', '0FC0', '0FC0', '0FF0',
47:          '0FF0', '0F80', '7F80', '7B00', '7300',
48:          '6380', '03C0' ) );
49:
50:      StuffHex( @frames[4],
51:        Concat( '0000', '0030', '0078', '00FC', '03FF',
52:          '00F8', '00FC', '00F8', '0070', '00E0',
53:          '01E0', '03E0', '03E0', '07E0', '07E0',
54:          '07F0', '07F0', '0FC0', '0FC0', '07C0',
55:          '0600', '0F00' ) );
56:
57:      StuffHex( @frames[5],
58:        Concat( '0000', '0030', '0078', '00FC', '03FF',
59:          '00F8', '00FC', '00F8', '0070', '03E0',
60:          '07E0', '07E0', '07E4', '07FC', '07FC',
61:          '03C0', '03E0', '0360', '0770', '1E78',
62:          '1C00', '1E00' ) );
63:
64:  END; { PrepareFrames }
65:
66:
67:  PROCEDURE InitDestRect;
68:
69:      { Set destRect to area where animation is to appear.  Assumes
70:        srcBits record initialized. }
71:
72:  BEGIN
73:      destRect := srcBits.bounds;
74:      OffsetRect( destRect, 50, 50 )
75:  END; { InitDestRect }

```

```

76:
77:
78:  PROCEDURE InitBitMap;
79:
80:  { Initialize global bit map record }
81:
82:  BEGIN
83:    WITH srcBits DO
84:      BEGIN
85:        baseAddr := NIL; { Filled in by CopyFrame }
86:        rowBytes := 2;
87:        SetRect( bounds, 0, 0, 16, maxRows )
88:      END; { with }
89:      InitDestRect
90:    END; { InitBitMap }
91:
92:
93:  PROCEDURE CopyFrame( frameNumber : INTEGER );
94:
95:  { Copy one bit map from off-screen memory to the display.
96:    Similar to displaying one frame of a cartoon film. }
97:
98:  BEGIN
99:    srcBits.baseAddr := @frames[ frameNumber ];
100:    CopyBits( srcBits, screenBits,
101:      srcBits.bounds, destRect, notSrcCopy, NIL );
102:    IF destRect.right < screenBits.bounds.right
103:    THEN
104:      BEGIN
105:        OffsetRect( destRect, 1, 0 ); { Move frame }
106:      END
107:    ELSE
108:      BEGIN
109:        FillRect( destRect, black ); { Reset to beginning }
110:        InitDestRect
111:      END
112:    END; { CopyFrame }
113:
114:
115:  PROCEDURE Pause( n : INTEGER );
116:
117:  { Wait for a small amount of time proportional to n }
118:
119:  BEGIN
120:    WHILE n > 0 DO n := n - 1
121:    END; { Pause }
122:
123:
124:  BEGIN
125:    PrepareFrames;
126:    InitBitMap;
127:    frameNumber := 1;
128:    REPEAT
129:      CopyFrame( frameNumber );
130:      Pause(10000);
131:      frameNumber := frameNumber + 1;
132:      IF frameNumber > maxframeNumber
133:      THEN frameNumber := 1
134:      UNTIL GetNextEvent( KeyDownMask, theEvent )
135:    END; { DoGraphics }

```

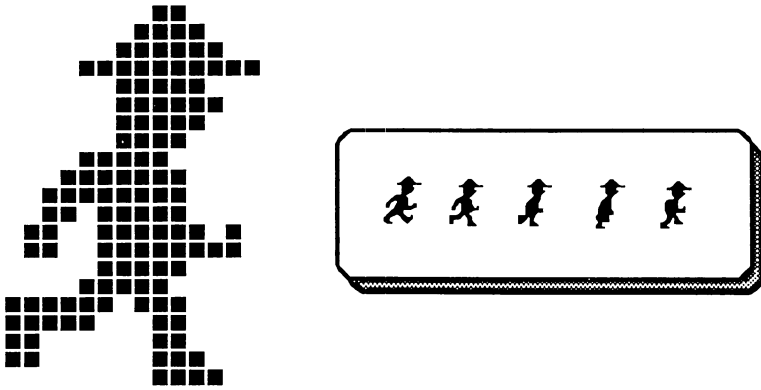



Figure 3.21 The blowup on the left is the second frame of five in the animation sequence on the right. Listing 3.6 displays each of these frames in rapid succession to animate a walking figure.

Animate Play-by-Play

When you run the program, you see a tiny figure (looking like a forest ranger to me) walking from left to right across the screen. As you can see, the animation is very smooth with no flicker. By following a few simple rules, you can do the same in your own programs.

Figure 3.21 is a blowup of the second animation frame of five, shown in the rounded box to the right. I drew these images with MacPaint. To make the blowup, I copied the FatBits screen to disk, then reloaded that image into MacPaint, cut the exploded figure, and copied it into the final picture along with the normal-sized frames. To design your own images, use a similar technique or fill in the squares on a sheet of graph paper. Keep your images small. Sixteen bits wide by 20 to 30 bits tall is an ideal size.

After designing the animation frames, the next step is to convert them to a form you can type in a program. Because each dot in an image equals a single memory bit, it's convenient to express bit images as hexadecimal digits 0 through F, representing the four-bit binary values 0000 through 1111. Figure 3.22 is a form you can use to convert your images to hex values. Each square represents a single bit in memory. The numbers along the top are the positions of each bit in a byte, with two bytes per row. These numbers and those on the left are for your reference only—they are not coordinate values. (If you purchased the disks that accompany this book, the form is in MacDraw file named *Form* in folder *Graphics.F*.)

	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	Hex	
0																	0 = 0000	
1																	1 = 0001	
2																	2 = 0010	
3																	3 = 0011	
4																	4 = 0100	
5																	5 = 0101	
6																	6 = 0110	
7																	7 = 0111	
8																	8 = 1000	
9																	9 = 1001	
10																	A = 1010	
11																	B = 1011	
12																	C = 1100	
13																	D = 1101	
14																	E = 1110	
15																	F = 1111	
16																		
17																		
18																		
19																		
20																		
21																		
22																		
23																		
24																		
25																		
26																		
27																		
28																		
29																		
30																		
31																		

Figure 3.22 Use this form to design your own animation frames. After filling in the boxes on the left (containing 32 rows of 16 cells each), convert groups of four cells into hex digits, writing them into the four-cell Hex column on the right. Use the hex-to-binary chart as a guide. A 1 stands for filled-in cells; 0 for blanks.

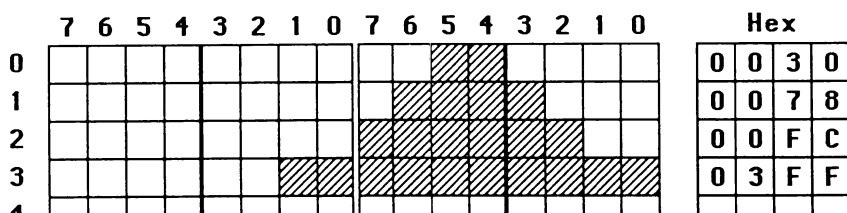


Figure 3.23 This shows the first four rows of the animation figure's hat in the form from Figure 3.22. The hex digits on the right represent the filled-in cells on the left as binary values.

PrepareFrames (23–64)

After X-ing in your image into Figure 3.22, use the hex table on the right to convert each group of four bits to one hex digit. To illustrate the process, Figure 3.23 shows the first four lines (the little fellow's hat) of the blowup frame from Figure 3.21 as you would enter them into the grid. The hex digits to the right represent the darkened squares to the left as 16-bit binary values.

With all the images designed and converted to hex digits, you're ready to type them into the program. Lines 29–34 in procedure **PrepareFrames** illustrate how to type each of the five frames. Procedure **StuffHex** takes a pointer to a variable (**@frame[1]**, for example) and a string of hex digits. When the program runs, it converts those digits into binary values and stuffs them four-bit nybble-by-nybble into memory. For clarity, **Concat** joins each group of four hex digits corresponding with the graph (Figures 3.22 and 3.23). But you could simply string them all together like this if you want: '00000030007800FC . . .'

Global array **frames** (20) holds five arrays of 22 integers, taking exactly 44 bytes of memory. One danger when using **StuffHex** to store binary values into variables is that it does not check whether the variable has enough room to hold the digits you stuff into it. Grouping values into four-digit strings as in the Listing makes it easy to count values and helps prevent overstuffing. Each four-digit group occupies 16 bits and, as you can see, there are 22 groups in each of the five **StuffHex** statements—exactly the size of each frame.

InitDestRect, InitBitMaps (67–90)

To animate images requires a destination rectangle to tell QuickDraw where to draw on screen. Procedure **InitDestRect** (67–75) sets global **destRect** to the same size as the **bitmap srcBit's bounds** rectangle. The **srcBit bitmap** points to the off-screen animation images stored in array **frames**. Using its **bounds** rectangle as the destination tells QuickDraw not only where to display images but also how large to draw them, in this case the same size as the original.

InitBitMap (78–90) defines the global **srcBits** record. It sets **baseAddr** to **NIL**, meaning nowhere in particular. For each animation frame, the base address of the bit image changes to one of the five images stored in array **frames**. The program uses the *same* bit map to define the location and size of each frame. Later, to draw different images, it needs only to assign **baseAddr** to the address of one image. It's important to understand that the **bitmap** record merely defines where in memory the image exists—it doesn't hold the image itself. Many people confuse the term bit map with the **bitmap** record. The bit map image contains the image; the **bit-map** record defines its location and size, a subtle and potentially confusing distinction.

Lines 86–87 finish defining **srcBits**, setting **rowBytes** to 2 (the number of bytes in one image row as Figure 3.23 shows), and calling **SetRect** to initialize the **bounds** rectangle. Fields **left** and **top** are usually zero as they are here. **Right** is 16 and **bottom** is equal to **maxRows**, the coordinates that exactly correspond with a single frame 16-pixels wide by 22-pixels tall. As its last job, **InitBitMap** calls **InitDestRect** (89) to set the global destination rectangle to the same size as the **srcBits.bounds** field and to position this rectangle somewhere on the display.

CopyFrame (93–112)

CopyFrame displays a single animation frame. By calling it repeatedly and cycling parameter **frameNumber** from one to five, the program draws successive animation frames at the **destRect** location. At the same time, moving **destRect** makes the animated figure move, appearing to walk from side to side.

Line 99 assigns the address of one frame to **srcBits baseAddr** field. Next, **CopyBits** copies that image to the area **destRect** defines on the visible screen. **CopyBits** has the general form:

```
PROCEDURE CopyBits (
    srcBits, dstBits : BitMap;
    srcRect, dstRect : Rect;
    mode : INTEGER;
    maskRgn : RgnHandle );
```

The first two parameters define the source **BitMap** (**srcBits**) and the destination (**dstBits**). In this example, the source is the off-screen animation frame; the destination is the display, a typical setup. Next are the source and destination rectangles, **srcRect** and **dstRect**. The example uses **srcBits.bounds** as the source rectangle to copy the entire animation frame to the display. In other situations, you could use a smaller source rectangle to display only a portion of the source bit image. The destination rectangle is **destRect**, the on-screen area where **CopyBits** transfers images. Parameter **mode** is any one of source transfer mode constants on the left side of Table 3.9. (See page 93.) Animations almost always use **srcCopy**

or, as in this example, **notSrcCopy** to display a white image on a black background. This causes each successive frame to completely overlay the previous image. To limit copying to a specific area, define a region and pass its handle to **CopyBits** as the last parameter. If you do that, images appear only inside the region's boundaries. The example passes **NIL** (101) for this value, telling **CopyBits** not to limit drawing to any particular boundaries.

All of this accomplishes one job—drawing a single animation frame. When done, lines 102–111 check whether **destRect** is at the extreme right edge. If not, line 105 advances the destination rectangle one pixel to the right, causing the figure to move. Turn line 105 into a comment, and the little fellow walks in place—he'll never reach the right edge.

Lines 109–111 erase the final image and reset the destination rectangle by calling **InitDestRect**. This causes the image to reappear at the left after bumping into the right screen border. If the program didn't do this, the little guy would walk to the ends of the earth or, rather, the end of the coordinate plane.

Pause, DoGraphics (115–135)

Pause (115–121) waits for a time proportional to parameter **n**, which has no relation to real time. Pass larger values to **Pause** to wait for longer times. The animation example calls it from line 130 with a value of 10,000, causing a 10,000-loop wait at line 120 between *each* animation frame. To see why this is necessary, turn line 130 into a comment and the little guy now runs an Olympian 100-yard dash. Try other values at line 130 to change animation speed, pausing for more or less time between frames.

The main **DoGraphics** loop (124–135) is simple. It first initializes the bit map and frame images (125–126) before setting **frameNumber** to one. The REPEAT loop (128–134) cycles variable **frameNumber** from 1 to 5 (**maxFrameNumber**), calling **CopyFrame** (129) to animate the display. The loop ends at line 134 when you press the space bar (or any other key).

You can adjust the animation by varying some of the parameters the program passes to **CopyBits**. For example, to make the figure walk down and to the right, change line 105 to:

```
OffsetRect( destRect, 1, 1 );
```

This adds one to both the horizontal and vertical coordinates of the image, making it travel diagonally. Another interesting trick is to alter not only the position of the **destRect** rectangle but also its size. Doing this causes QuickDraw to *scale* the original image up or down to match the new destination boundaries. To make the figure grow larger, as though it were walking toward you, add the following statement between lines 105 and 106:

```
IF frameNumber = maxFrameNumber
  THEN InsetRect( destRect, -1, -1 );
```

The negative values expand **destRect** by one pixel along each border. (Positive values shrink **destRect**.) The **IF** statement prevents the image from growing too fast, changing size only just before starting a new animation sequence.

Another modification you can try is to insert **WHILE** loops to wait for mouse clicks between animation frames. This is useful when you want to study each image in slow motion. I used the idea when designing the example here in order to refine the images. Add these statements between lines 129 and 130, just after the call to **CopyFrame**. (If you later have trouble ending the program, hold down the mouse button while you press the space bar. That should work.) You need two **WHILE** statements because you probably cannot click the mouse quickly enough to avoid advancing the image beyond one frame at a time. Take out the second **WHILE** statement and you'll see what I mean.

```
WHILE NOT Button DO {wait for button};
WHILE Button DO {wait for release};
```

Avoiding Animation Flicker

As you can see, Listing 3.6 smoothly animates the figure without the flicker you may have seen in other programs. It accomplishes this magic with the help of a simple trick.

Many programmers attempt animation by drawing an image, erasing it, and then redrawing it a tiny distance away. Unfortunately, the erase step causes an objectionable flicker, which requires careful planning to avoid. For a demonstration, change two of the four-digit hex values in lines 33 and 34 as follows:

```
'70F8' change to 'F0F8'
'7070' change to 'F070'
```

When you run the modified program, the figure leaves a part of his shoe behind as he walks from left to right. This happens because the modified values insert single bits in the extreme left column of the image. (Hex 7 is 0111 in binary; F is 1111 and has an extra 1 bit to the left.)

By leaving the far left column blank, moving the image one pixel to the right causes it to erase itself as it walks. Similarly, leaving a blank row on top lets the image move one pixel down. This may be difficult to visualize, but it always works. To prove to yourself that it does, sketch figures into the worksheet in Figure 3.22. Leave the left column blank. Hold successive frames up to the light and shift them as they would during animation. As you can see with this experiment, subsequent frames always erase any far left bits in previous images. Without the blank column, bits are not erased, leaving a trail. This leads to a simple rule for designing animation frames that erase themselves as they travel, avoiding flicker.

Leave blank columns or rows in the opposite direction of travel as many pixels wide as the number of pixels the image shifts during animation.

FRACTALS

The final example in this chapter demonstrates several more QuickDraw features with a program that grows life-like patterns such as those in Figure 3.24. To me, these shapes resemble sea coral, moss, or nerve complexes. They grow as the result of a simple idea, explained by Leonard M. Sander in “Fractal Growth,”

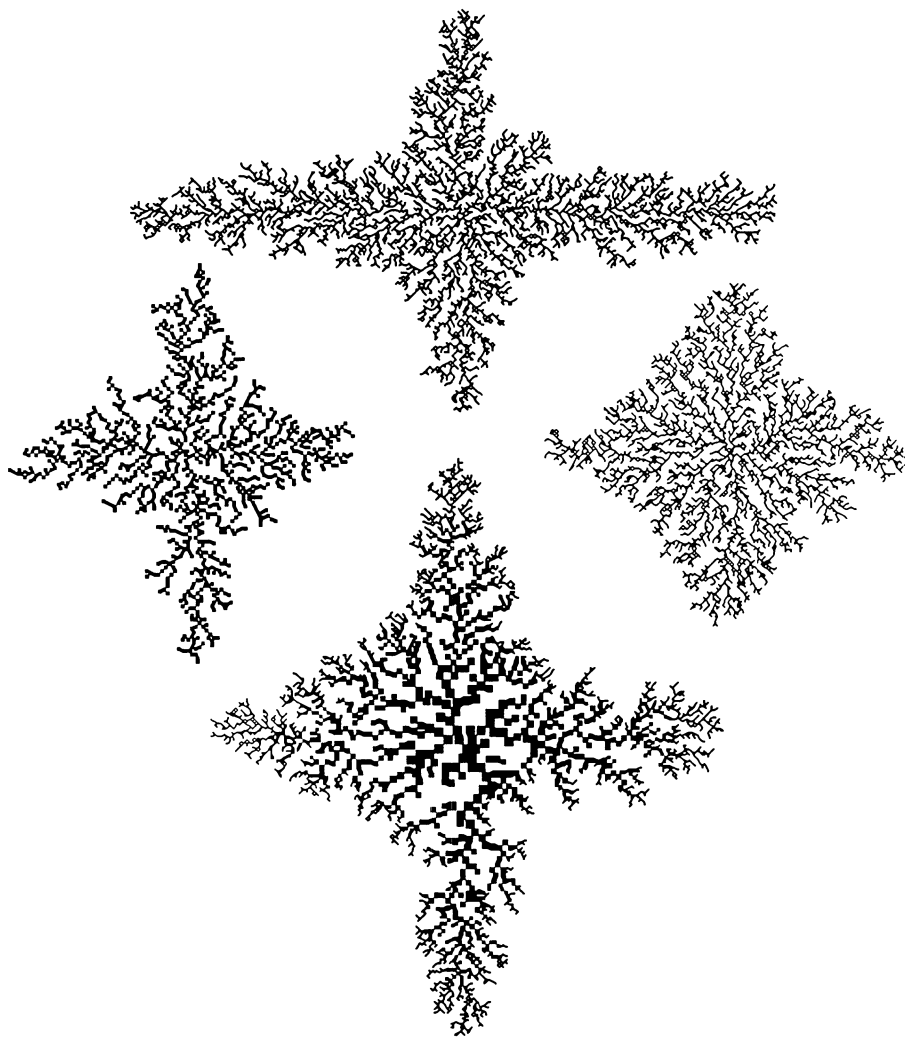


Figure 3.24 Variations of Listing 3.7 drew these four fractal-like images a pixel at a time. Even though the Macintosh is a fast computer, it took many hours to produce these pictures.

Scientific American, January 1987, Vol. 256, No. 1, p. 94. But the reason they look as they do remains a mystery, at least in a mathematical sense.

A typical fractal image resembles a shoreline. From high above the shore, you see the general outline of continents with many bays and peninsulas. As you descend, you see less of the shoreline but more detail. When very close, perhaps on your hands and knees and looking through a magnifier, you still see something that looks like a shoreline. No matter how close you get, the shoreline retains its general characteristics.

The following program draws fractals of a different sort. In this case, objects grow by releasing small dots somewhere outside of the visible display. The program walks these dots toward the center, letting them wander at random as they travel. A single dot at center serves as a seed to which the wandering dots attach. When a moving dot touches another, it sticks to it, perhaps resembling the way the skeletons of tiny marine polyps congregate to form coral.

Interestingly, that simple idea produced the four patterns in Figure 3.24. In the bottom pattern, the dots vary in size with later ones becoming smaller as the shape grows. As you can see, the individual feathers are similar to the larger stems—they exhibit a fractal-like appearance. The theory is that the shapes grow because randomly traveling dots are more likely to stick to bumps, forming larger bumps which are more likely to attract more dots, and so on. This tends to amplify small irregularities, producing regular patterns from random events.

Type in Listing 3.7 and save as FRACTAL.PAS. Plan to let this program run for a long time—it took several *hours* to produce the patterns in the figure.

Listing 3.7. FRACTAL.PAS

```

1: {$O Programs:Graphics.F: }      { Send compiled code to here }
2: {$U-}                           { Turn off standard library units }
3:
4:
5: PROGRAM Fractal;
6:
7: (*
8:
9:  * PURPOSE : Fractal graphics
10:  * SYSTEM  : Macintosh / Turbo Pascal
11:  * AUTHOR   : Tom Swan
12:
13:  * Based on ideas appearing in "Fractal Growth" by Leonard M. Sander;
14:  * Scientific American, Jan 1987, Vol 256, No 1, pg 94
15:
16:  *)
17:
18:
19:  USES
20:
21:      Memtypes, QuickDraw, OSIntf, ToolIntf;
22:
23:
24:  VAR
25:
26:      gPort : GrafPort;
27:
28:

```

(continued)


```

29: FUNCTION Quitting : BOOLEAN;
30:
31: { TRUE if next event is a MouseDown event }
32:
33:     VAR
34:
35:         event : EventRecord;
36:
37:     BEGIN
38:         IF GetNextEvent( everyEvent, event )
39:             THEN Quitting := ( event.what = MouseDown )
40:             ELSE Quitting := FALSE
41:         END; { Quitting }
42:
43:
44: PROCEDURE Randomize;
45:
46: { Start new random sequence }
47:
48:     VAR
49:
50:         time : LONGINT;
51:
52:     BEGIN
53:         GetDateTime( time );
54:         RandSeed := time
55:     END; { Randomize }
56:
57:
58: PROCEDURE SetupScreen;
59:
60: { Initialize display for graphics }
61:
62:     BEGIN
63:         OpenPort( @gPort );           { Open new graphics port }
64:         PenPat( Black );              { Select drawing color }
65:         PaintRect( gPort.portRect )   { Fill screen with black }
66:     END; { SetupScreen }
67:
68:
69: PROCEDURE Plot( h, v : INTEGER );
70:
71: { Plot a single white point at coordinate h,v }
72:
73:     BEGIN
74:         MoveTo( h, v );
75:         PenPat( White );
76:         LineTo( h, v );
77:         PenPat( Black )
78:     END; { Plot }
79:
80:
81: PROCEDURE UnPlot( h, v : INTEGER );
82:
83: { Plot a single black point (erasing a white point there) at h,v }
84:
85:     BEGIN
86:         MoveTo( h, v );
87:         PenPat( Black );
88:         LineTo( h, v )
89:     END; { Plot }
90:
91:
92: PROCEDURE DoGraphics;
93:
94: { Create display }
95:

```

```

96:  CONST
97:
98:      MaxPS      = 16; { Maximum dimensions of a single point }
99:      ps         = 2;  { Actual pen size 2 <= ps <= MaxPS }
100:     WalkSpeed   = 2;  { Positive values >= 2 to change speed }
101:     DecisionSpeed = 24; { Lower values for "crazier" movement }
102:
103:
104:  VAR
105:
106:      pointSet    : SET OF Byte; { Set of pixels in one point }
107:      hMax        : INTEGER;      { Maximum h coordinate value }
108:      vMax        : INTEGER;      { Maximum v coordinate value }
109:      radius      : REAL;         { Circle radius for new points }
110:      leftBorder  : INTEGER;      { Drawing border coordinates }
111:      rightBorder : INTEGER;
112:      topBorder   : INTEGER;
113:      bottomBorder : INTEGER;
114:
115:
116:  PROCEDURE Initialize;
117:
118:  { Initialize global variables and change origin }
119:
120:  VAR
121:
122:      r      : Rect;
123:      x, y   : REAL;
124:
125:  BEGIN
126:
127:      Randomize; { Start new random sequence }
128:
129:      WITH screenBits.bounds DO
130:      BEGIN
131:          hMax := right; { Calculate maximum horizontal }
132:          vMax := bottom; { and vertical dimensions. }
133:          rightBorder := ( right DIV 2 ) - 1; { Calculate border }
134:          bottomBorder := ( bottom DIV 2 ) - 1; { coordinates. }
135:          leftBorder := -1 * rightBorder;
136:          topBorder := -1 * bottomBorder
137:      END; { with }
138:
139:
140:  { Set origin so that (0,0) is at screen center }
141:
142:  SetOrigin( -1 * ( hMax DIV 2 ), -1 * ( vMax DIV 2 ) );
143:
144:
145:  { Calculate radius of a circle that encloses the display. The }
146:  { radius is equal to the length of a line from the display center }
147:  { to one of its corners. }
148:
149:  x := hMax / 2.0;
150:  y := vMax / 2.0;
151:  radius := sqrt( x * x + y * y );
152:
153:
154:  ClipRect( gPort.portRect ); { Clip to visible area }
155:  PenSize( ps, ps ); { Preset pensize for each point }
156:  pointSet := [ 0 .. ps-1 ]; { The set of pixels in one point }
157:  Plot( 0, 0 ) { Start fractal with single "seed" }
158:
159:  END; { Initialize }
160:
161:

```

(continued)

```

162:  PROCEDURE NewDelta( n : INTEGER; VAR d : INTEGER );
163:
164:  { Return new direction and speed value in d, so that }
165:  { n tends to converge toward 0 }
166:
167:      VAR
168:
169:          j : INTEGER;
170:
171:      BEGIN
172:          j := ABS( Random ) MOD WalkSpeed;
173:          IF n > 0
174:              THEN d := -j
175:              ELSE d := j
176:          END; { NewDelta }
177:
178:
179:  PROCEDURE Walk( h, v : INTEGER; VAR dh, dv, nH, nV : INTEGER );
180:
181:  { Adjust coordinate h,v producing new coordinate nH,nV and making }
182:  { the original point tend to walk randomly toward the origin 0,0 }
183:
184:      BEGIN
185:          IF Random MOD DecisionSpeed = 0 { i.e. once in a while... }
186:              THEN NewDelta( h, dh );      { ... change directions }
187:          IF Random MOD DecisionSpeed = 0
188:              THEN NewDelta( v, dv );
189:          nH := h + dh; { Move point by delta h and v }
190:          nV := v + dv
191:          END; { Walk }
192:
193:
194:  PROCEDURE StartNewPoint( VAR h, v, dh, dv : INTEGER );
195:
196:  { Start new point outside a circle enclosing entire screen }
197:
198:      VAR
199:
200:          angle : INTEGER;
201:          w      : REAL;
202:
203:      BEGIN
204:          angle := ABS( Random ) MOD 360; { Random value from 0 to 359 }
205:          w := angle * Pi / 180.0;        { w = angle in radians }
206:          h := TRUNC( radius * cos(w) );  { Calculate coordinate h,v }
207:          v := TRUNC( radius * sin(w) );  { on circle's circumference. }
208:          NewDelta( h, dh );              { Initialize speed and }
209:          NewDelta( v, dv )               { direction values. }
210:          END; { StartNewPoint }
211:
212:
213:  FUNCTION PixelOn( h, v : INTEGER ) : BOOLEAN;
214:
215:  { TRUE if pixel at h,v is white and inside borders }
216:
217:      BEGIN
218:          PixelOn := FALSE;
219:          IF GetPixel( h, v ) THEN exit ELSE
220:          IF ( h <= leftBorder ) OR ( h >= rightBorder ) THEN exit ELSE
221:          IF ( v <= topBorder ) OR ( v >= bottomBorder ) THEN exit;
222:          PixelOn := TRUE
223:          END; { PixelOn }
224:
225:

```

```

226: FUNCTION Stuck( h, v : INTEGER ) : BOOLEAN;
227:
228: { TRUE if another point borders the one at h,v }
229:
230: VAR
231:
232:     i, j : INTEGER;
233:
234: BEGIN
235:     FOR i := -1 TO ps DO
236:         FOR j := -1 TO ps DO
237:             IF PixelOn( h + i, v + j ) THEN
238:                 IF NOT ( ( i IN pointSet ) AND
239:                     ( j IN pointSet ) ) THEN
240:                     BEGIN
241:                         Stuck := TRUE;
242:                         exit
243:                     END;
244:                     Stuck := FALSE
245:                 END; { Stuck }
246:
247:
248: PROCEDURE RandomWalk;
249:
250: { Make a point walk randomly toward center and stick to any }
251: { existing points. }
252:
253: VAR
254:
255:     pH, pV, npH, npV, dh, dv : INTEGER;    { point coordinates }
256:
257: BEGIN
258:     StartNewPoint( pH, pV, dh, dv );        { Get new point values }
259:     Plot( pH, pV );                          { Display initial position }
260:     WHILE ( NOT Stuck( pH, pV ) ) AND        { Do following until stuck }
261:         ( NOT Button ) DO                    { to a point or mouse down }
262:         BEGIN
263:             Walk( pH, pV, dh, dv, npH, npV ); { Move point (maybe) }
264:             UnPlot( pH, pV );                 { Erase old position }
265:             Plot( npH, npV );                 { Display new position }
266:             pH := npH;                        { Remember values for }
267:             pV := npV;                        { next possible loop }
268:         END { while }
269:     END; { RandomWalk }
270:
271:
272: BEGIN
273:     Initialize;
274:     REPEAT
275:         RandomWalk
276:     UNTIL Quitting
277: END; { DoGraphics }
278:
279:
280: BEGIN
281:     InitGraf( @thePort );                    { Initialize Quickdraw }
282:     InitCursor;                               { Make sure cursor level = 0 }
283:     HideCursor;                              { Make cursor invisible }
284:     FlushEvents( everyEvent, 0 );            { Erase any pending events }
285:     SetupScreen;                             { Prepare display for graphics }
286:     DoGraphics
287: END.

```

Fractal Play-by-Play

Much of the program is probably familiar by now. Function **Quitting** (29–41) returns **TRUE** when you click the mouse. This lets you write loops such as at lines 274–276 later on in the program. One advantage of using this function instead of **Button** to sense mouse clicks as in earlier examples is the ability to type Command-Shift-3 to copy the display to MacPaint disk files. This works because line 38 calls **GetNextEvent**, giving the operating system the opportunity to sense your command to save the screen to disk.

Plot, UnPlot (69–89)

These two little procedures belong in every graphics toolbox. **Plot** paints a single dot equal to the pen width and height at coordinate (h,v). Before **Plot** ends, it changes the pen to **Black** (77). For black on white displays, you might want to change it to **White** instead.

UnPlot (81–89) reverses what **Plot** does. It paints a single dot in black, erasing a white dot at this position. Fractal calls **UnPlot** to erase dots before moving them in order to show the Fractal's individual elements as they form. (This also produces a slight animation flicker avoided in the previous example. But, because Fractal animates only single dots, the flicker is hardly noticeable. It would be if you drew larger images this way.)

DoGraphics (92–277)

DoGraphics is a long procedure that contains several others. First come four constants (98–101) that you can change to produce different effects. **MaxPS** is here as a reminder that individual dots probably should not be larger than 16 pixels square. The constant isn't used anywhere in the program. Constant **ps** defines the pen size, with the width always equal to its height. Large values (4–8 or higher, but less than **MaxPS**) produce blocky figures as in the center of the bottom fractal in Figure 3.24. Setting **ps** to 1 produces fine hair-like structures as in the top of the figure. You might turn **ps** into an integer variable and vary it randomly or change it over a period of time. I used a similar approach (not listed here) to produce the bottom fractal.

Constant **WalkSpeed** affects the amount dots move during each program cycle. Larger values move dots more quickly but initially makes it harder for them to stick together. Values greater than 8 or so are probably too large. Try 3 or 4.

DecisionSpeed controls the behavior of dot movement by varying the amount of time it takes for a dot to “decide” to turn. Larger values make dots plod forward in more or less straight lines. Lower values make dots go crazy, like fruit flies over a rotten banana.

Variable **pointSet** (106) stores a set of coordinate values that belong to a single

dot of any size. The program uses **pointSet** while examining its border to determine if the dot has touched another. (See lines 238–239.) Variables **hMax** and **vMax** hold the display maximum and minimum coordinates, which the program gets from global **screenBits** so that Fractal works on any size display.

Variable **radius** defines an imaginary circle that completely encloses the rectangular display. Fractal always releases new dots on this circle's circumference. You might consider changing this algorithm—release dots along one or two borders, or from a single location—and see what effect such changes have.

The four integer variables, **leftBorder** to **bottomBorder** (110–113) help determine whether dots are inside or outside the visible display. They speed the program by avoiding repeated references to the fields in the **screenBits.bounds** rectangle, an action that takes Pascal longer to calculate than referring directly to simple variables.

Initialize (116–159)

Procedure **Initialize** has plenty to do. It first starts a new random sequence (127) and then assigns values to global variables (131–136). It also adjusts border variables to place coordinate (0,0) in screen center. Line 142 completes this idea by calling **SetOrigin**, which has the general form:

```
SetOrigin( h, v : INTEGER );
```

Passing (0,0) to **SetOrigin** gives the pixel in the top left corner that coordinate, the default condition when you initialize QuickDraw. Passing negative values moves the (0,0) coordinate down and to the right—just what you would expect if the pixel in the top left corner had negative coordinate values. Passing negative values equal to one half the screen width and height, as in line 142, shifts the origin to the screen center.

Shifting the origin doesn't move images now on display—it affects only what you later draw there. If you draw a line and then change the origin, the line does not move. But if you draw a second line at the same coordinates, it appears at a different location after the origin changes.

To understand what **SetOrigin** does, imagine the display as a window through which you view a portion of the coordinate plane (see Figure 3.25). Shifting the origin moves the window over the plane, exposing different areas. Drawing occurs not on the plane, but on the window glass, sticking to it so that, when you shift its location by calling **SetOrigin**, anything already on display moves along. (In other words, what's now on display appears *not* to move from your perspective.)

After shifting the origin to center coordinate (0,0), **Initialize** calculates the radius of a circle that encompasses the entire display (149–151). Because new dots start on that circle's circumference, line 154 sets clipping to the visible display rectangle, limiting drawing to that area. Line 155 sets the pen size. After that, line 156 ini-

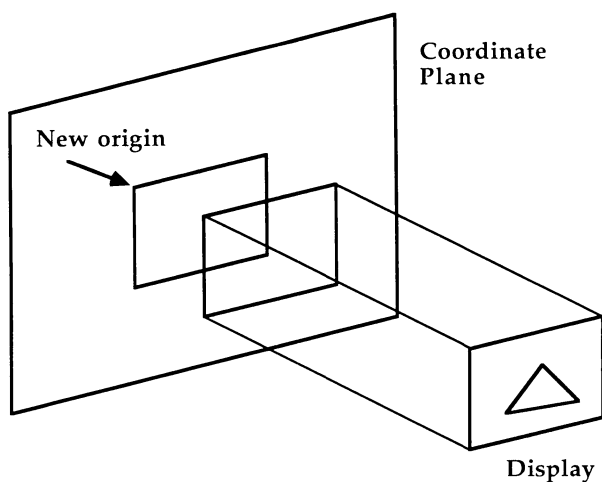


Figure 3.25 When shifting the origin—telling QuickDraw to view a different portion of the entire coordinate plane—existing images *stick* to the display and do not move. For an example, see Listing 3.7 which shifts the origin placing coordinate (0,0) at screen center.

tializes **pointSet** to the set of all points in a pen of that size. And finally, line 157 seeds the image, placing a starting point in the center of the display. (Notice that if you did not shift the origin, line 157 would display a point in the upper left corner.)

NewDelta, Walk (162–191)

Together, **NewDelta** and **Walk** move dots at random around the display. The dots tend to converge toward the center, sticking to other dots they touch. **Walk** calls **NewDelta** after using constant **DecisionSpeed** to determine whether to change a dot's direction (185–188). Larger **DecisionSpeed** values cause the expressions in 185 and 187 to produce zero less often. Therefore, the dots change directions more infrequently. Lower values do the opposite.

NewDelta returns a positive value at random limited to the **WalkSpeed** less one (172). By examining the dot's current coordinate value **n** and setting the new direction negative if **n** is greater than zero or positive if not, **NewDelta** ensures that points try to move toward the center—without forcing them to travel directly there. In theory, a dot might wander forever and never approach the center. But that's not likely. At least it hasn't yet happened to me.

After possibly setting new direction values in **dh** and **dv**, **Walk** adds those values to the current coordinate, passing the result to the caller in parameters **nH** and **nV** (189–90). This is the direction in which the current dot then moves.

StartNewPoint (194–210)

This procedure decides where the next point should begin. It also randomly sets the direction of the point by calling **NewDelta** twice (208–209). Lines 204–207 calculate a coordinate (h,v) on the circumference of an imaginary circle with a large radius. It does this by selecting an angle at random between zero and 359 degrees (204) and converting that angle to the radians (205) that Pascal's **COS** (Cosine) and **SIN** (Sine) functions require. Lines 206–207 use standard geometry to locate the endpoint (h,v) of an imaginary line starting from the circle's center and pointing out at this angle.

All of this may seem overly complex simply to start new dots. But placing them on the circle's circumference seems to give each dot an equal shot at reaching the center. With that in mind, you might try other methods for releasing new dots. What happens, for example, if you start them purely at random on the entire coordinate plane or on the borders of a triangle?

PixelOn, Stuck (213–245)

PixelOn returns **TRUE** if the pixel at coordinate (h,v) is white and inside the visible display borders. The function calls QuickDraw's **GetPixel** function, which returns **TRUE** if the point at (h,v) is black. (Remember, black pixels are normally *on* (1); white ones are *off* (0). The graphics programs in this chapter reverse that logic and, therefore, cannot directly use **GetPixel**'s result.) Lines 220–221 solve an apparent quirk in **GetPixel**, which returns **FALSE** for pixels outside the screen boundaries. Checking that (h,v) is inside the screen borders solves this problem.

Function **Stuck** (226–245) returns **TRUE** if the dot at coordinate (h,v) touches another already on display. It does this by examining every pixel around the dot's border, calling **PixelOn** to look for adjacent dots.

RandomWalk, DoGraphics (248–277)

RandomWalk controls the dot action, calling **StartNewPoint**, **Plot**, **Walk**, and **UnPlot** to move dots toward the center, stopping only when a dot sticks to another or when you press the mouse button. The main procedure loop (272–277) simply initializes the program and calls **RandomWalk** repeatedly until you quit.

In Any Event

The Macintosh is an eventful computer in more than just a casual sense. As you probably know, there's plenty of action and interaction in Macintosh software—pull-down menus, desk accessories running along with other programs, and windows that you can move and resize by clicking and dragging the mouse. To make all of these actions cooperate, programs respond to *events*—mouse clicks, requests to redraw window contents, and other things—rather than issue *commands* as in conventional software design.

Because of the *event-driven* nature of a Macintosh program, routines often appear out of place. Suppose someone requests an action, perhaps to draw a figure inside a window. Instead of the program doing that, it might change a variable or two but not actually draw anything at that time. Or it might call a procedure that collects information about items that require updating. Later, another routine actually draws the figure based on this information. Rather than drawing directly in windows, programs tell the toolbox that something needs changing in a window, and the toolbox issues an event that eventually ends up calling your drawing routine. How to write such event-driven software is the subject of this chapter.

It may seem overly complex to write programs by the event-driven method. Why *not* simply draw things in windows? Isn't that easier? Yes, but in event-driven systems, foreign programs can issue events to which your program must respond. (An example of a foreign program is a desk accessory or a device driver in memory.) If all you want to do is draw figures in windows, then you can write Turtle Graphics or simple QuickDraw programs similar to the examples in previous chapters. But adding pull-down menus, movable windows, and desk accessories requires the fancy footwork that event-driven software allows.

In this chapter, you'll learn about the basic parts of Pascal programs that follow event-driven programming rules. You'll be able to write programs to activate the Macintosh pull-down menu bar, add overlapping windows, and use desk accessories. The goal of this chapter is to develop an application shell, an empty vessel into which you insert your own routines, similar to the graphics shell in the previous chapter. The shell handles most of the details of an event-driven program letting

you concentrate on other jobs rather than forcing you to rewrite the same procedures over every time you start a new program. Most of the remaining examples in this book use variations of the shell in this chapter.

THE PARTS OF AN APPLICATION

Based on thousands of programming lines, public domain examples, and documents released by Apple Computer to software developers, it appears that most event-driven programs have six fundamental parts:

1. Global Declarations
2. Program Actions
3. Display Handlers
4. Event Handlers
5. Initializations
6. Program Engine

It will help you to write your own event-driven programs if you understand the nature of these six parts. Dividing your program this way helps reduce common confusions in event-driven programming where actions seem not to take place at logical times but, rather, in procedures that appear to have nothing to do with the actions you want to perform. The best way to avoid such confusion is to concentrate on writing code for each part's *purpose*, rather than concentrating on your routine's *order* among the other programming statements. Event-driven programming is purposeful programming. You concentrate more on purpose than arrangement.

The following notes describe the six parts to an application. If you look ahead to Listing 4.1 on page 129, you'll see these same parts as large comment blocks. For example, see lines 26 and 54. Even if you don't understand all the programming in the listing, you might want to glance at it while you read the following description about the way an event-driven program works.

Global Declarations

A program's Global Declarations define the constants, data types, and variables that the program uses. Although this is no different than in normal Pascal, every Macintosh program begins with a list of declarations usually describing, among other things, the identifying numbers of resources that the program uses. (A resource is a predefined object such as a template for a window or a menu title. You create resources separately and tell Turbo to combine them with your program to produce the finished result.)

Because the Macintosh memory manager limits you to about 32K for all global

variables, it's probably best to be frugal in your declarations. Don't create large arrays and other data structures that take a lot of memory. If you do, you might run out of room. This doesn't mean programs cannot have large variables. They can if you create them on the heap at run-time, a technique I'll show you as we proceed.

Program Actions

Program Actions handle whatever unique operations the program does. In a printing program, you'd put the printing routines in this part. In a database, you'd add searching and sorting procedures, and so on.

Program Actions include procedures to respond to pull-down menu choices. They also include programming to respond to clicking the mouse in a window or to pressing the Return key. Understand the difference here between the *response* to something and the *sensing* of the event that requires such a response. Sensing a mouse click in a window's close box and actually closing the window are different though related processes. In the Program Actions section, concentrate on what it is you want mouse clicks and keypresses to do—not on the method by which your program knows whether mouse clicks have occurred. Understand this distinction and apply it when writing your own programs. It's vital to good Macintosh software design.

Display Handlers

Display Handlers are responsible for drawing shapes, figures, lines, text, and other graphics, usually in windows. Similar to a Program Action, a Display Handler concerns itself with *what* it should do—never with *when* it should do it. For example, in a program that displays a bar graph, you would write a Display Handler routine to display graphs in windows. But you would not worry about when bar graphs should appear, and you would not be concerned with the commands people might give in order to display the graphs.

You might wonder how this works if, for example, you have to draw a *different* bar graph in response to a command or an option setting of some kind. Suppose you need to draw an oval at one time but a box at another depending upon which of those shapes someone tells the program to draw. In such circumstances, it seems impossible to disassociate completely your drawing routine from the actions that require it to respond differently at one time or another.

But you can easily handle these and other instances where window contents change at different times. One solution is to write a Program Action routine to set variables for specific commands and a Display Handler to examine those variables in order to know whether to draw one object or another. As a very simple example, the following procedures demonstrate this idea. First comes the global variable and Program Action procedure:

```

VAR
    whichFigure : ( anOval , aBox );

PROCEDURE DoCommand( command : INTEGER );
BEGIN
    IF command = 0
        THEN whichFigure := anOval
        ELSE whichFigure := aBox
    END; { DoCommand }

```

DoCommand simply sets **whichFigure** to **anOval** or **aBox** depending on the value of its parameter, **command**. A zero specifies an oval; other values specify a **box**. Exactly how **command** gets its value is unimportant. What matters is the fact that **DoCommand** itself doesn't draw any ovals or boxes. It merely changes the program's *knowledge*. As a Program Action, it responds to commands. It's the Display Handler's job to actually draw the appropriate figure. There, you might use this routine:

```

PROCEDURE DrawContents;

VAR
    r : Rect;

BEGIN
    SetRect( r , 10 , 10 , 75 , 75 );
    IF whichFigure = anOval
        THEN FrameOval( r )
        ELSE FrameRect( r )
    END; { DrawContents }

```

DrawContents draws an oval with QuickDraw's **FrameOval** procedure or a box with **FrameRect** depending on the value of the global variable, **whichFigure**. Be certain you understand the relationship between **DoCommand** (the Program Action procedure) and **DrawContents** (the Display Handler). **DoCommand** assigns a value to **whichFigure**. **DrawContents** examines that value to know which figure to draw—either an oval or a box. **DoCommand** knows nothing about drawing figures. **DrawContents** knows nothing about program commands. Each does its job and neither knows anything about the other nor about why, when, or how it might be called to respond as the program runs.

Event Handlers

Event Handlers direct program flow. They have complete responsibility for calling Program Actions and Display Handlers according to a program's needs along with the needs of other processes that run at the same time. Remember always that

your program is rarely alone. Desk accessories and devices such as disk drives and serial input ports have their own needs to which your program must respond. For example, if you move a desk accessory window to one side, your program must redraw the newly exposed portion of a window underneath. If you click an inactive window, your program must deactivate the current window, bring the new window to the front and, in the process, draw the contents of the now frontmost window. Event Handlers receive these and other events such as mouse clicks and keypresses. They determine the nature of the events and call the appropriate Program Action or Display Handler in response. In general, there are only four main events that you need to handle in most programs. These are:

- Mouse down events
- Key down events
- Update events
- Activate events

Mouse down events occur every time you click the mouse button. Because the mouse operates independently (you can almost always move the mouse pointer, even while other operations proceed), mouse down events can occur at any time. The mouse down Event Handler checks the location of the mouse pointer and, based on that location, determines what other procedures to call in order to respond to mouse clicks. If, for example, you choose a menu command, the Event Handler calls a Program Action procedure to respond. If you click inside an inactive window, the Event Handler brings that window to the front.

Key down events occur when you type keys. To respond, the Event Handler checks whether you also held down the command key to choose a menu command. If so, it calls the appropriate Program Action to respond to the command—the same way it does for mouse down events that choose commands from pull-down menus. It handles normal keypresses by calling a Program Action that presumably knows what to do with typing—maybe inserting a character into a text document or simulating cursor keys.

Update events take place as the result of actions that require redrawing items on display. In response, the program activates your Display Handler routine, drawing the contents of one or another window. For example, the program might receive an update event when you change a window's size, uncovering in the process the contents of another window below. Or, you might close a window, erasing it from the display. The update routine would take care of calling your Display Handler to redraw the contents of any windows previously hidden from view.

Activate events are the fourth type. They occur when windows become active or inactive in response to mouse clicks and to commands that create new windows overtop those already on display. (Despite its name, an activate event can be either for making windows active or for making them inactive. The activate Event Handler takes care of both kinds of activate events.) Most of the time, activate events and update events come in pairs. When you click inside an inactive window, it becomes

active, requiring the program to draw any newly visible parts and also deactivating a previously active window. It's rare that you'll need to know it but, in such cases, deactivate events come first—they have priority over activate events.

Initializations

Every application needs to initialize its variables before the program kicks into high gear. On the Macintosh, initializations prepare pull-down menus and call certain toolbox routines to allow them to set up their own variables.

Usually, initialization procedures run only one time at the start of the program. Some initializations you must do. Others are optional. Still others you determine according to the needs of your program.

For lack of a better place, I include among an application's Initializations a *shut-down* procedure—a routine that programs call just before they end. (You might think of it as a deinitialization procedure.) There's been much written about how to start a Macintosh program but not as much about properly ending one. Because there might be open windows, changed documents, and other unfinished business when someone chooses the Quit command, it's important for programs to respond properly. The best way to do this is with a deinitialization plan that guards against losing information if you end too early, perhaps forgetting to save changed data on display in a window.

Program Engine

The final part of a standard Macintosh application is the Program Engine, the motor that makes a program go. Its first job is to call the program's Initialization part. Then, as all engines, it cycles—in this case, repeating these important jobs for as long as the program runs:

- Perform system operations
- Intercept events
- Ensure a proper shut-down

Perform system operations. The Program Engine repeatedly calls procedure **SystemTask**, which gives desk accessories such as the alarm clock a chance to update their displays. (In long loops, call **SystemTask** if you know the Program Engine will be idle for more than 1/60 second—not a hard and fast rule, but one to observe as closely as possible.) Along with calling **SystemTask**, the Engine dims or highlights certain menu commands depending on which windows are active. It can also change cursor shapes depending on the mouse pointer's location. In the coming program shell, procedure **DoSystemTask** performs these system operations.

Intercept events. Most important of the Program Engine's responsibilities is its job of intercepting and directing events. When you click the mouse button, the Macintosh operating system posts an event, meaning it inserts into a special list, or queue, a data object that records where the mouse was (among other things) at the time you clicked its button. The Program Engine obtains this plus other events by repeatedly calling Boolean function **GetNextEvent**, which returns **TRUE** if it can pass back an event of a type the Engine requests. (It's possible to request the next event of a particular kind—only mouse clicks, for example—and ignore others. Usually, though, the Program Engine requests all events and simply ignores the ones it doesn't care about.)

After receiving an event, the Program Engine directs it to one of the four Event Handlers, calling procedures to handle mouse down, key down, update, and activate events as they occur. This drives the program, putting into effect all its other parts at the proper times. Update events find their way to Display Handler procedures while mouse down events cause Program Actions to respond to commands. These actions are at the heart of an event-driven program—driven by the cycling Program Engine.

Ensure a proper shut-down. The final job ends programs in a logical, clearly defined way following the idea that program actions should not have direct effects but, rather, should change a program's knowledge. In this case, the action might be what happens when you choose the pull-down menu's Quit command. The knowledge is that fact—we'll call it **QuitRequested**. If **QuitRequested** is **TRUE**, then the program assumes that you want it to stop. (It doesn't care how or why it acquired this knowledge.) Sensing this, the Program Engine verifies that it is indeed okay to end the program at this time. If not—for example, if you added text to a document but forgot to save those changes—it calls the appropriate procedures to let you clean up before ending. This also gives you the chance to change your mind and not quit the program after all. Putting these ideas together, the Program Engine has the following Pascal-like form:

```
BEGIN
  Initialize;
  REPEAT
    Do system tasks;
    Direct events to Event Handlers;
  UNTIL QuitConfirmed
END.
```

If you are familiar with Macintosh programming, you know that this Engine—more commonly called the main loop—is simpler than usual. It initializes the program and then cycles, performing system tasks and directing events to Event Handler routines. Very important is the next to last line where Boolean function **QuitConfirmed** checks whether it's time to end the program and, if all is well, returns **TRUE**, giving the Engine permission to turn itself off.

That concludes the description of a Macintosh application. It's not vital for you to memorize every preceding point but, rather, to have a good sense of what the term event-driven means. Before continuing, you should understand that the actions a program takes are separate from the ordering of those actions. You should understand that the Program Engine directs events to Event Handlers, which take care of calling the appropriate routines to perform program chores.

The rest of this chapter develops the programming steps to implement a full-bodied Macintosh application shell that fleshes out these six basic parts. Although meaty, it's still mostly a skeleton that doesn't do any useful processing. Even so, the shell is invaluable for writing new programs. Rather than starting from scratch, you begin with a copy of the bare-bones skeleton. This saves time while ensuring that you don't forget an important step.

DEVELOPING AN APPLICATION—APSHELL

Together, Listings 4.1, 4.2, and 4.3 make up an application shell, AP-SHELL.PAS, that fully implements the six parts of a Macintosh event-driven program. In many cases, you can write complete programs by replacing only one or two shell procedures and adding your own constants and variables. Chapters 5–7 have many such examples.

The first ApShell part, Listing 4.1, contains routines and other declarations that might change from one program to another. Its many *place holders* are procedures that show you where to add your own programming. The second ApShell part, Listing 4.2, contains the text form of the shell's resources. It describes the menus, English language strings, and a window template that goes with the shell. The third ApShell part is a unit, MacExtras in Listing 4.3, which contains many useful tools that probably won't change for different programs.

ApShell offers several features. It adds desk accessories to the pull-down menu bar and lets you close them either by clicking their close boxes or by choosing the Close command from the File menu. Many programs (including Turbo Pascal) fail to follow this recommended guideline—Close should work for any frontmost window, whether or not it belongs to your program.

The shell also has a complete Edit menu with Undo, Cut, Copy, Paste, and Clear commands that work correctly with desk accessories such as the familiar Note Pad, Key Caps, and Scrapbook utilities. Unlike many programs, ApShell properly dims and highlights this menu at appropriate times rather than allow you to select an Edit command even when the program doesn't allow cutting and pasting.

Another shell feature is *window zooming*. By clicking the zoom box in the upper right corner of a window, you expand the window to full-screen size. Clicking the zoom box again shrinks the window to its former size. (Window zooming is available only on Macintoshes with 128K ROMs. Older 64K ROM systems can use the shell without modification but won't have zoomable windows.)

ApShell has one more feature not often included in other program shells: a mechanism to help you properly shut down a program, closing opened windows

and saving data without having to write a series of Boolean flags or fiddle with the event queue. Of course, in the shell itself, there isn't anything to save. But later, in programs that create disk files, you'll see how this mechanism protects you from losing your work.

The following sections describe each of the three listings that make up the shell. If you want to type in the entire program now, skip the Play-by-Play descriptions but read the introductions that precede each listing. They contain notes about compiling the listings to produce a finished program.

Listing 4.1 is the main ApShell listing. Type it in and save as APSHELL.PAS. Change lines 1 and 2 to use different volume and folder names. You cannot run ApShell until you type in the other two listings (4.2 and 4.3). Continue to the next listing under the heading ApShell Resources if you want to type in the entire program before reading the play-by-play descriptions.

Listing 4.1. APSHELL.PAS

```

1: {$O Programs:Shells.F: }           { Send compiled code to here }
2: {$R Programs:Shells.F:ApShell.Rsrc} { Use this compiled resource file }
3: {$U-}                               { Turn off standard library units }
4:
5:
6: PROGRAM ApShell;
7:
8: (*
9:
10:  * PURPOSE : Application shell
11:  * SYSTEM  : Macintosh / Turbo Pascal
12:  * AUTHOR  : Tom Swan
13:
14: *)
15:
16:
17: {$U Programs:Units.F:MacExtras }    { Open this library unit file }
18:
19:
20:  USES
21:
22:      Memtypes, QuickDraw, OSIntf, ToolIntf, PackIntf, MacExtras;
23:
24:
25:
26: { ----- }
27: { }
28: {           G L O B A L   D E C L A R A T I O N S           }
29: { }
30: { ----- }
31:
32:  CONST
33:
34:      FileID      = 2;      { File menu Resource ID and commands }
35:      NewCmd       = 1;
36:      CloseCmd     = 2;
37:      {-----}
38:      QuitCmd      = 4;
39:
40:      WindowID     = 1;     { Window resource ID }
41:
42:
43:

```

(continued)

```

44:   VAR
45:
46:   wRec           : WindowRecord;      { Program's window data record }
47:   wPtr           : WindowPtr;        { Pointer to above wRec }
48:
49:   quitRequested  : BOOLEAN;          { TRUE if quitting }
50:   windowOpen     : BOOLEAN;          { TRUE only if window is open }
51:
52:
53:
54: { ----- }
55: {
56:           P R O G R A M   A C T I O N S
57: }
58: { ----- }
59:
60:
61: PROCEDURE DoKeyPress( ch : CHAR );
62:
63: { Do something with an incoming character }
64:
65:   BEGIN
66:   END; { DoKeyPress }
67:
68:
69: PROCEDURE DoMouseClicked( whichWindow : WindowPtr );
70:
71: { Process mouse clicks inside windows }
72:
73:   BEGIN
74:   END; { DoMouseClicked }
75:
76:
77: PROCEDURE DoNew;
78:
79: { Respond to File menu New command }
80:
81:   BEGIN
82:     IF NOT windowOpen THEN
83:       BEGIN
84:         wPtr := GetNewWindow( WindowID, @Wrec, POINTER( -1 ) );
85:         windowOpen := wPtr <> NIL;
86:         IF windowOpen THEN
87:           BEGIN
88:             SetPort( wPtr );
89:             EnableItem( fileMenu, CloseCmd );
90:             DisableItem( fileMenu, NewCmd )
91:           END
92:         END { if }
93:       END; { DoNew }
94:
95:
96: PROCEDURE CloseProgramWindow;
97:
98: { Close the global wPtr window }
99:
100:   BEGIN
101:
102:     IF windowOpen THEN
103:       BEGIN
104:         CloseWindow( wPtr );
105:         windowOpen := FALSE;
106:         EnableItem( fileMenu, NewCmd );
107:         DisableItem( fileMenu, CloseCmd )
108:       END { if }
109:
110:     END; { CloseProgramWindow }
111:

```

```

112:
113: PROCEDURE DoClose;
114:
115: { Respond to File menu Close command }
116:
117: BEGIN
118:     IF FrontWindow = wPtr
119:     THEN CloseProgramWindow    { Close the program's window }
120:     ELSE CloseDAWindow        { Close desk accessory window }
121:     END; { DoClose }
122:
123:
124: PROCEDURE DoFileMenuCommands( cmdNumber : INTEGER );
125:
126: { Execute command in the File menu }
127:
128: BEGIN
129:     CASE cmdNumber OF
130:         NewCmd      : DoNew;
131:         CloseCmd    : DoClose;
132:         QuitCmd     : quitRequested := TRUE
133:     END { case }
134:     END; { DoFileMenuCommands }
135:
136:
137: PROCEDURE DoEditMenuCommands( cmdNumber : INTEGER );
138:
139: { Execute command in the Edit menu }
140:
141: BEGIN
142:     IF NOT SystemEdit( cmdNumber - 1 ) THEN
143:     BEGIN
144:         { Do program's edit menu commands }
145:     END { if }
146:     END; { DoEditMenuCommands }
147:
148:
149: PROCEDURE DoCommand( command : LONGINT );
150:
151: { Execute a menu command }
152:
153: VAR
154:
155:     whichMenu  : INTEGER;    { Menu number of selected command }
156:     whichItem  : INTEGER;    { Menu item number of command }
157:
158: BEGIN
159:
160:     whichMenu := HiWord( command );    { Find the menu }
161:     whichItem := LoWord( command );    { Find the item }
162:
163:     CASE whichMenu OF
164:
165:         AppleID  : DoAppleMenuCommands( whichItem );
166:         FileID   : DoFileMenuCommands(  whichItem );
167:         EditID   : DoEditMenuCommands(  whichItem );
168:
169:         { Add other program menus here }
170:
171:     END; { case }
172:
173:     HiliteMenu( 0 ) { Unhighlight menu title }
174:
175:     END; { DoCommand }
176:
177:
178:

```

(continued)

132 *≡ Programming with Macintosh Turbo Pascal*

```
179: { ----- }
180: { }
181: { D I S P L A Y   H A N D L E R S }
182: { }
183: { ----- }
184:
185:
186: PROCEDURE DrawScrollBars( whichWindow : WindowPtr );
187:
188: { Draw v & h scroll bars.  In the shell, this draws only the scroll
189:   bar outline and grow box. }
190:
191:   VAR
192:
193:     vBarRect    : Rect;      { Vertical scroll bar }
194:     hBarRect    : Rect;      { Horizontal scroll bar }
195:     gbRect      : Rect;      { Grow box }
196:
197:   BEGIN
198:     DrawGrowIcon( whichWindow );
199:     CalcControlRects( whichWindow, hBarRect, vBarRect, gbRect );
200:     ValidRect( hBarRect );
201:     ValidRect( vBarRect );
202:     ValidRect( gbRect );
203:   END; { DrawScrollBars }
204:
205:
206: PROCEDURE DrawContents( whichWindow : WindowPtr );
207:
208: { Display window contents }
209:
210:   BEGIN
211:     EraseRect( whichWindow^.portRect );
212:     DrawScrollBars( whichWindow );
213:
214:     { Add commands to draw window's contents }
215:
216:   END; { DrawContents }
217:
218:
219:
220: { ----- }
221: { }
222: { E V E N T   H A N D L E R S }
223: { }
224: { ----- }
225:
226:
227: PROCEDURE MouseDownEvents;
228:
229: { Someone pressed the mouse button.  Check its location and respond. }
230:
231:   VAR
232:
233:     partCode : INTEGER;      { Identifies what item was clicked. }
234:
235:   BEGIN
236:
237:     WITH theEvent DO
238:
239:     BEGIN
240:
241:       partCode := FindWindow( where, whichWindow );
242:
243:       CASE partCode OF
244:
```

```

245:         inMenuBar
246:             : DoCommand( MenuSelect( where ) );
247:
248:         inSysWindow
249:             : SystemClick( theEvent, whichWindow );
250:
251:         inContent
252:             : IF whichWindow <> FrontWindow
253:                 THEN SelectWindow( whichWindow )
254:                 ELSE DoMouseClicked( whichWindow );
255:
256:         inDrag
257:             : DragTheWindow( whichWindow, where );
258:
259:         inGrow
260:             : IF whichWindow <> FrontWindow
261:                 THEN SelectWindow( whichWindow )
262:                 ELSE ResizeWindow( whichWindow, theEvent.where );
263:
264:         inGoAway
265:             : IF TrackGoAway( whichWindow, where )
266:                 THEN DoClose;
267:
268:         inZoomIn, inZoomOut
269:             : IF TrackBox( whichWindow, where, partCode )
270:                 THEN ZoomInOut( whichWindow, partCode )
271:
272:     END { case }
273:
274:     END { with }
275:
276: END; { MouseDownEvents }
277:
278:
279: PROCEDURE KeyDownEvents;
280:
281: { A key was pressed. Do something with incoming character. }
282:
283:     VAR
284:
285:         ch : CHAR;
286:
287:     BEGIN
288:         WITH theEvent DO
289:             BEGIN
290:
291:                 ch := CHR( BitAnd( message, charCodeMask ) ); { Get character }
292:
293:                 IF BitAnd( modifiers, CmdKey ) <> 0 { If command key pressed }
294:                     THEN DoCommand( MenuKey( ch ) ) { then execute command }
295:                     ELSE DoKeypress( ch ) { else use character }
296:
297:             END { with }
298:         END; { KeyDownEvents }
299:
300:
301: PROCEDURE UpdateEvents;
302:
303: { Part or all of a window requires redrawing }
304:
305:     VAR
306:
307:         oldPort : GrafPtr; { For saving / restoring port }
308:

```

(continued)

```

309: BEGIN
310:   GetPort( oldPort );           { Save current port }
311:   whichWindow :=
312:     WindowPtr( theEvent.message ); { Extract window pointer }
313:   SetPort( whichWindow );       { Change current grafPort }
314:   BeginUpdate( whichWindow );   { Calculate new visRgn }
315:   DrawContents( whichWindow );  { Draw/redraw window contents }
316:   EndUpdate( whichWindow );     { Reset original visRgn }
317:   SetPort( oldPort )           { Restore old port }
318: END; { UpdateEvents }
319:
320:
321: PROCEDURE ActivateEvents;
322:
323: { Activate or deactivate windows }
324:
325: BEGIN
326:   WITH theEvent DO
327:     BEGIN
328:
329:       whichWindow := WindowPtr( message ); { Extract window pointer }
330:       SetPort( whichWindow );             { Change current port }
331:
332:       DrawScrollBars( whichWindow );      { Draw bars & grow box }
333:
334:       IF BitAnd( modifiers, activeFlag ) <> 0
335:       THEN FixEditMenu( FALSE )          { Activate a window }
336:       ELSE FixEditMenu( TRUE )           { Deactivate a window }
337:
338:     END { with }
339:   END; { ActivateEvents }
340:
341:
342:
343: { ----- }
344: {
345: {           I N I T I A L I Z A T I O N S
346: {
347: { ----- }
348:
349:
350: PROCEDURE SetUpMenuBar;
351:
352: { Initialize and display menu bar }
353:
354: BEGIN
355:
356:   appleMenu := GetMenu( AppleID ); { Read menu resources }
357:   fileMenu  := GetMenu( FileID );
358:   editMenu  := GetMenu( EditID );
359:
360:   InsertMenu( appleMenu, 0 );      { Insert into menu list }
361:   InsertMenu( fileMenu, 0 );
362:   InsertMenu( editMenu, 0 );
363:
364:   AddResMenu( appleMenu, 'DRVR' ); { Add desk accessory names }
365:
366:   DrawMenuBar                      { Display the menu bar }
367:
368: END; { SetUpMenuBar }
369:
370:
371: PROCEDURE Initialize;
372:
373: { Program calls this routine one time at start }
374:

```

```

375: BEGIN
376:     SetUpMenuBar;           { Initialize and display menus }
377:     quitRequested := FALSE;  { TRUE on selecting Quit command }
378:     windowOpen := FALSE     { TRUE after using New command }
379: END; { Initialize }
380:
381:
382: FUNCTION QuitConfirmed : BOOLEAN;
383:
384: { The program's "deinitialization" routine. If someone chooses quit
385: command, this routine closes any open windows and tells the main
386: program loop whether it is okay to end the program now. }
387:
388: BEGIN
389:     IF quitRequested THEN
390:         IF windowOpen
391:             THEN CloseProgramWindow;
392:         QuitConfirmed := quitRequested
393:     END; { QuitConfirmed }
394:
395:
396:
397: { ----- }
398: {
399: {               P R O G R A M   E N G I N E
400: {
401: { ----- }
402:
403:
404: PROCEDURE DoSystemTasks;
405:
406: { Do operations at each pass through main program loop }
407:
408: BEGIN
409:
410:     SystemTask;    { Give DAs their fair share of time }
411:
412:     IF FrontWindow = NIL THEN
413:
414:         BEGIN { Set up menu commands for empty desktop }
415:
416:             FixEditMenu( FALSE );
417:             EnableItem( fileMenu, NewCmd );
418:             DisableItem( fileMenu, CloseCmd );
419:
420:         END ELSE
421:
422:         IF FrontWindow <> wPtr THEN
423:
424:             BEGIN { Set up menu commands for active desk accessory }
425:
426:                 FixEditMenu( TRUE );
427:                 EnableItem( fileMenu, CloseCmd );
428:
429:             END { else / if }
430:
431:         END; { DoSystemTasks }
432:
433:
434: BEGIN
435:
436:     Initialize;
437:
438:     REPEAT
439:
440:         DoSystemTasks;
441:

```

(continued)


```

442:      IF GetNextEvent( everyEvent, theEvent ) THEN
443:
444:          CASE theEvent.what OF
445:
446:              MouseDown      : MouseDownEvents;
447:              KeyDown        : KeyDownEvents;
448:              AutoKey         : { ignored };
449:              UpdateEvt       : UpdateEvents;
450:              ActivateEvt     : ActivateEvents
451:
452:          END { case }
453:
454:      UNTIL QuitConfirmed
455:
456:  END.

```

ApShell Play-by-Play

Lines 1–3 select three compiler directives that most of the remaining programs in this book use. Line 1 you’ve seen before. It tells Turbo where to send the output code file when it compiles this program to disk. It has no effect on compiling to memory.

Line 2 tells the compiler which resource file to combine with this program’s code. (This file contains the binary form of the program’s resources, not the text in Listing 4.2. Don’t confuse the two.) When Turbo compiles a program to memory, it opens the file in line 2 to read its resources as needed while your program runs. When compiling to disk, it combines the resources with the program code, producing a complete application in a single file. This adds menu titles, window templates, and other resources to programs.

Line 3 switches off Turbo’s standard library units, eliminating its dumb terminal interface. Because we want to write programs that open their own windows, add desk accessories, and contain pull-down menus, we don’t want Turbo’s fixed window to interfere. Unfortunately, switching off standard library units has another effect that makes programming more difficult. No longer can you add a simple **Writeln** statement to display text in a window. Without the standard units in effect, you have to use QuickDraw commands to *draw* text. (You can also use the Macintosh TextEdit tools to display text—but more on that later.) The advantage, of course, is that you can use the many available fonts and styles to display text. And you can combine graphics and text on screen in any way you can imagine.

It’s helpful to know exactly what you’ve turned off in line 3. With standard library units on { \$U + }, Turbo automatically includes three units with every program it compiles: PasSystem, PasInOut, and PasConsole. PasSystem adds low-level items such as math routines, string handlers, sets, and other native Pascal elements. PasInOut adds standard I/O routines to programs. This includes **Write**, **Writeln**, **Read**, and **Readln** along with code to implement files that you use with those procedures to read and write data to disk or to devices like printers and modems. PasConsole contains the dumb terminal interface that you use in textbook programs.

Of these three, Turbo always adds PasSystem. You cannot remove Pascal's fundamental abilities to use sets and handle strings (nor would you want to). Turning off the standards eliminates only the two units, PasInOut and PasConsole.

Having switched off standard units, you can always explicitly add them back in the program's USES clause. For example, a useful debugging technique is to change line 20 to read:

```
USES PasInOut, PasPrinter,
```

and then insert statements to print various information while a program runs. Let's say you have a variable **XMax** that you suspect is not being initialized properly. Somewhere in the program, you can write:

```
WRITELN( PRINTER, 'XMAX=', XMAX );
```

When the program gets to that statement, it prints **Xmax**'s value. **PRINTER** is defined in the PasPrinter unit. Adding it as the first parameter in **Write** and **Writeln** procedures sends text and other items to the printer. Another trick is to put statements like these at the beginning and end of procedures.

```
PROCEDURE A;
BEGIN
    WRITELN( PRINTER, 'ENTER PROCEDURE A' );

    . . .

    WRITELN( PRINTER, 'EXIT PROCEDURE A' )
END;
```

Doing this in every procedure, or in a select few, traces a program's execution, listing all the procedures it calls. You might also print variables to determine if they contain what you think they should. Printing data this way is a useful debugging technique and, because you remain on Pascal's level, is often more useful than tracing the machine language code with a conventional debugger. As shown here, I usually write my debugging procedures in all uppercase to make them easy to find and remove later.

One unit you should never use with standard units switched off is PasConsole, which initializes Turbo's dumb terminal and interferes with Macintosh programs that set up their own windows. Never insert PasConsole in a program's USES clause. Let Pascal automatically include it when you compile textbook programs (without the {SU-} directive in line 3).

Notice that line 17 tells Pascal to read the compiled MacExtras unit (Listing 4.3). Be sure to modify this line if you compile MacExtras to different volume and folder names. Alternatively, you can use Turbo's UnitMover program to add MacExtras to the compiler. In that case, remove line 17. (See Chapter 7 for details on using UnitMover.)

Lines 20–22 tell Pascal to use six units, including MacExtras. The newcomer here is PackIntf (Package Manager Interface), which adds packages to your program. Originally, a package was a kind of after-thought toolset—containing programming that, for one reason or another, was left out of the Macintosh ROMs. Today, packages are just miscellaneous toolsets in the System file (called RAM-based tools because they share memory with your program's code) or in ROM. One package adds the standard file dialog for selecting file names, ejecting disks, and opening folders. Another standardizes date and time formats and selects international currency symbols and decimal points depending on where in the world your program is running. By using the International Utilities Package, your program can automatically use pound signs instead of dollar symbols and commas instead of periods for decimal points—strange to us Yankees perhaps, but all the same to a program in London.

ApShell Global Declarations (26–50)

As in all Macintosh applications, ApShell has a Global Declaration part. Constants define the File menu and a window. If you've already run the shell, you might wonder where the Apple and Edit menus are. Because these rarely change, their constants are in the MacExtras unit (Listing 4.3). Even though ApShell doesn't define them, you certainly can add commands to these menus. Future programs explain how.

The five constants in lines 34–38 require explanation. **FileID** is the identifier that the Macintosh Resource Manager toolset uses to read this menu into memory. The menu resource comes from the resource file, which you specify in line 2 and which you create by running Listing 4.2 through RMaker, the Resource Compiler program that comes with your Turbo Pascal system. At line 39 in Listing 4.2, you'll see the resource ID 2, the same value as in line 34 of ApShell.

The File menu's commands (35–38) are constants, too, but their values do not represent resource IDs. Instead, a menu command's value is simply its position in the menu. The first command is always 1, the second 2, and so on. Observe one caution when assigning values to these constants. As line 37 shows, you must allow for divider lines in menus even though such lines are deadwood—they don't do anything, they just help organize menus into subcategories. As far as the Menu Manager toolset is concerned, though, a divider line still is a command even though no one can choose it. Notice that QuitCmd's value is 4, not 3, leaving that value for the divider line, represented in the listing by the comment {-----}.

Line 40 in ApShell defines another constant, **WindowID**. Similar to **FileID**, this is the number of a corresponding resource definition. In this case, **WindowID** refers to a window *template*, which describes the size, location, and style of windows that the program uses. The template is the resource that defines what the window looks like, not the window data structure as it exists in memory. The line that

corresponds to the constant definition is in Listing 4.2 at line 67. The shell uses **WindowID** to load this template into memory when it creates a new window.

Lines 46–47 declare two variables that add a single window to the shell: **wRec** keeps track of various details such as the window's location, its style and features (such as whether it has a go-away box); **wPtr** (47) points to the **wRec** variable. This may seem odd to expert Pascal programmers who are familiar with the normal use of pointers, which rarely if ever address common variables. In Turbo Pascal, this is not only possible but critical to Macintosh programming.

Despite its name, window pointer **wPtr** does not point to a structure of type **WindowRecord**, the record in which the toolbox Window Manager keeps facts about a program's windows. (The *Guide* and *Inside Macintosh* describe the **WindowRecord** data type in full.) Window pointers actually point to **GrafPorts**, exactly the same records used in the graphics programs in Chapter 3. In other words, the definition for **WindowPtr** is:

```
TYPE
  WindowPtr = GrafPtr;
```

As you can see, there is no difference between **WindowPtr** and **GrafPtr** variables—both point to **GrafPort** records. The Macintosh toolbox accomplishes this apparent magic because every **WindowRecord** contains a **GrafPort** record as its first field. Because of this duality, you can pass the address of a window record to procedures that operate on windows or to procedures that operate on **GrafPorts**. For example, you can pass ApShell's **wPtr** to Window Manager procedure **SelectWindow**, which takes as its parameter a **WindowPtr** variable, or to **SetPort**, which takes a **GrafPtr** variable.

Still, even though **WindowPtr** and **GrafPtr** types are the same, and you can pass variables of either type to any procedure that requires the other, you can use only **WindowPtr** variables for windows and **GrafPtr** variables for pure **GrafPorts**. The reason for this is to prevent you from mixing the two types in cases where it does matter to which kind of record they point. Although they include a **GrafPort** as their first field, **WindowRecords** attach additional fields that the Window Manager requires in order to manipulate windows. For those rare times when you need to access those fields, Pascal defines another pointer type as follows:

```
TYPE
  WindowPeek = ^WindowRecord;
```

Never declare variables of this type. Except for low-level procedures for which you probably will have little use, no Window Manager procedures accept them as parameters. The way to use the **WindowPeek** data type is in a *type casting* statement, one that converts one data type to another by using the type identifier with a variable in parentheses. An example helps clarify how this works. Let's say you declare **wPtr** as in line 47 but later want to read the value of the **WindowRecord**

field **goAwayFlag**, a Boolean variable that tells whether this window has a go-away box in the upper left corner. You cannot do this:

```
IF wPtr^.goAwayFlag
  THEN {do something};
```

because **wPtr** points to a **GrafPort** and such records do not have **goAwayFlag** fields. Still, you know that **wPtr** actually addresses a **WindowRecord**—it's the compiler that doesn't have that same understanding. Therefore, to tell Pascal to ignore what it thinks it knows—to force it, in other words, to consider that **wPtr** addresses a window record—you recast the pointer into a new role, using **WindowPeek** this way:

```
IF WindowPeek( wPtr )^.goAwayFlag
  THEN {do something};
```

This use of **WindowPeek** generates no code—it just looks as though it does. It's not a function call, but a *translation* of a variable's data type (in this case a **WindowPtr**) into something else (**WindowPeek**). Notice that the caret, dereferencing the pointer, comes after the parentheses. If you put the caret inside after **wPtr**, you receive an “Invalid type cast argument,” meaning Pascal knows better than to allow you to convert an entire **GrafPort** into a **WindowPeek** variable. When type casting one type to another, they must have the same byte size. Other than that, there's no restriction.

The other two variables in ApShell's Global Declarations are two Boolean flags, **quitRequested** and **windowOpen** (49–50). The first, **quitRequested**, is **TRUE** after someone chooses the File menu's Quit command. This single flag controls the Program Engine's task of ensuring a secure shut-down before the program ends. The second Boolean variable, **windowOpen**, is **TRUE** if a program window is open. It does not indicate whether any other windows are open, such as those belonging to desk accessories. In a way, this flag is redundant. You could set **wPtr** to **NIL**—the pointer value that means “nowhere in particular”—to indicate that no window is open. But having the **windowOpen** flag makes for more readable programs. Instead of statements such as:

```
IF wPtr <> NIL
  THEN EraseWindow( wPtr );
```

you can write the more understandable:

```
IF windowOpen
  THEN EraseWindow( wPtr );
```

ApShell Program Actions (54–175)

The eight procedures in ApShell's Program Actions respond to commands that you choose from pull-down menus or by typing command keys like Command-Q.

Because they *Do* the things that make this program unique—in other words, because they perform the program's actions—procedure names in this section typically begin with “Do.” For example, the **DoNew** procedure performs the action you want when someone chooses the File menu's New command.

Two procedures, **DoKeyPress** and **DoMouseClicked** (61–74) are do-nothing shells—empty place holders that you fill in later. **DoKeyPress** receives a character typed on the keyboard. In the shell, typing has no effect and the procedure ignores the characters it receives. To prove that it works, replace it with the following programming. (Use a copy of ApShell to protect your original text. Never modify your only copy of the shell—you'll need it for future examples and for your own programs.)

```
PROCEDURE DoKeyPress( ch : CHAR );
BEGIN
  IF ch = '@'
    THEN quitRequested := TRUE
  END; { DoKeyPress }
```

When you run the modified program, type an at-sign (@) and the program ends. Notice that this happens without your knowing exactly the steps involved in shutting down the program. The procedure merely sets Boolean flag **quitRequested** to **TRUE** indicating that the state of the program has changed—that is, somebody typed the new at-sign Quit command.

Another do-nothing procedure, **DoMouseClicked** (69–74), takes care of mouse clicks in windows. Its parameter is a **WindowPtr** variable that addresses the window to which the mouse pointer points. Similar to the way you tested **DoKeyPress**, you can verify that **DoMouseClicked** works by replacing it with the following.

```
PROCEDURE DoMouseClicked( whichWindow : WindowPtr );
BEGIN
  SysBeep( 2 )
END; { DoMouseClicked }
```

SysBeep sounds a tone for the length of time in parentheses. The value stands for the number of ticks, or 1/60-second internal heartbeats in sync with Macintosh display updates, known technically as *vertical retrace interrupts*. Supposedly, the tone lasts for that length of time. But it's only an approximation of real time—the actual value is not that accurate.

When you run the program, choose the File menu's New command and click the mouse inside the window. Notice that it beeps only when the pointer touches the area under the window drag bar and inside the other borders.

By the way, remember **SysBeep** for those times when you simply want to know whether a certain procedure runs. Sometimes, you may wonder if a section of code executes. Insert a **SysBeep** and run the program to receive an audible answer.

DoNew (77-93)

Procedure **DoNew** opens a new window in response to choosing the File menu's New command. In a sense, this breaks the rule that action procedures affect only a program's knowledge. To follow that rule exactly, **DoNew** would have to generate an event that would *later* result in a window opening. Although not shown here, doing this requires creating your own custom event type and then adding code to intercept that event in the program's Event Handler section. There is an operating system function called **PostEvent** for this purpose but, in this case, using it would only complicate **DoNew** to no advantage.

Lines 82-92 check for open windows by examining the **windowOpen** flag. Because the shell dims the New command after opening a window, it shouldn't be possible to accidentally open another. Even so, checking the **windowOpen** flag eliminates the slightest possibility of an accident, a good rule of thumb to follow.

Line 84 creates and displays the window in a single statement. It does this by calling function **GetNewWindow**, which reads the window template from the resource file and returns a pointer to a new window record containing all the details that the Window Manager needs to manipulate this window. There are several important points to observe in line 84.

The first parameter, **WindowID**, is the constant with the resource ID value explained earlier. The second parameter, **@wRec**, equals the address of the global **wRec** variable. **GetNewWindow** needs this address to know where to do its work. The third and last parameter is **POINTER(-1)**, another example of type casting. In this case, the integer value **-1** is recast as a general **POINTER**, a generic type that Pascal recognizes as being compatible with any other kind of pointer. The value **(-1)** tells **GetNewWindow** to place the new window in front of all others. You can use **NIL** as the third parameter to create new windows *behind* all others on display. Or, you can pass the **WindowPtr** address of another window to create your new window behind that one. Despite these choices, you'll usually just use **POINTER(-1)** and create the new window on top. It's difficult to imagine a situation where you wouldn't want to do that.

The rest of **DoNew** checks to see if the call to **GetNewWindow** succeeded by testing in line 85 whether **wPtr** is **NIL**, in which case something went wrong trying to create the new window. As long as all is okay, lines 88-90 ensure that the program window is the current **GrafPort** (88), enable the File menu's Close command (89), and disable Open (90) to prevent opening another window. **EnableItem** and **DisableItem** are routines in the toolbox's Menu Manager. **EnableItem** activates a menu command, allowing you to choose it from a menu. **DisableItem** does the opposite, dimming the command and preventing you from choosing it.

You might wonder where variable **fileMenu** comes from. The MacExtras unit defines this and two other menus (**appleMenu** and **editMenu**), which are practically set-in-concrete standards in Macintosh software. The MacExtras play-by-play explains how to use them.

CloseProgramWindow, DoClose (96–121)

These two procedures go together. Only if a window is open does line 104 close it, removing it from the display and also erasing from memory certain miscellaneous structures that the Window Manager creates when you open a new window. **CloseWindow** is the correct procedure to use when your window record is on the stack or, as **wRec**, declared as a global variable. Another procedure, **DisposeWindow**, does the same job as **CloseWindow**, but also makes available the memory that the window record occupies. Never use **DisposeWindow** to close windows when the window record is a Pascal variable. Later, we'll see how to use this technique to manage window records on the heap, where you must be concerned with disposing objects you no longer need.

CloseProgramWindow also sets **windowOpen** to **FALSE** (105) and changes the File menu commands to the proper state when there aren't any open windows, dimming the Close and activating the New commands.

Procedure **DoClose** (113–121) checks if the front window belongs to the program. It does this by calling function **FrontWindow** (118), which returns a **WindowPtr** to the window now active. By checking whether this pointer is the same as the value in global variable **wPtr**, **DoClose** determines if the window belongs to a desk accessory or to this program. If it's a desk accessory window, **DoClose** calls **CloseDAWindow** in the MacExtras unit. Otherwise it calls **CloseProgramWindow** described earlier.

DoFileMenuCommands, DoEditCommands, and DoCommand (124–175)

The remaining three procedures in ApShell's Program Actions section are **DoFileMenuCommands**, **DoEditMenuCommands**, and **DoCommand**. Let's take the last one first, as it merely calls the others.

As you can see at lines 163–171, **DoCommand** directs a menu command to one of the Do . . . MenuCommands procedures that precede it in the program. **DoCommand**'s single **LONGINT** parameter, **command**, contains the menu ID number along with the line number of a command selected from a menu. The first job is to extract those parts, which **DoCommand** does at lines 160–161. Function **HiWord** returns the high-order 16-bits from a 32-bit **LONGINT** (see Figure 4.1). **LoWord** returns the low-order 16-bits.

Two local variables, **whichMenu** and **whichItem**, save the extractions of the **command** parameter. The first of these, **whichMenu**, figures in the **CASE** statement at lines 163–171. It selects one of the cases labeled by the menu resource ID numbers, **AppleID**, **FileID**, and **EditID**. If you have other menus, put their ID numbers after **EditID** where the comment indicates.

DoCommand passes **whichItem**, representing the menu command number, to one of the Do . . . MenuCommands procedures. Notice that **DoCommand** only

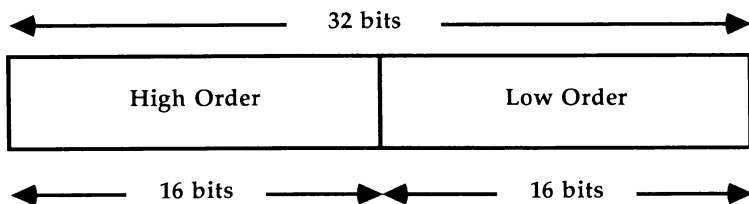


Figure 4.1 In memory, 32-bit objects order their bytes as shown here. The high order bits precede the low order—the opposite of the way some computers (the IBM PC for example) store multi-byte values.

redirects a command to the proper procedure—it doesn't take any actions itself. One of those procedures, **DoAppleMenuCommands**, is in the MacExtras unit. The Apple menu usually contains the familiar About Program command and desk accessories. Because it rarely varies from that setup, it's best kept in the unit along with other common routines.

DoCommand's final task is to call **HiliteMenu** (173), changing the now inverted menu title back to black on white, its normal state. When you choose a menu command, the Menu Manager erases the pull-down menu but leaves the menu title in reversed white on black. Surround **HiliteMenu(0)** with comment brackets to see what effect this has when you choose menu commands.

Backing up a few lines, **DoFileMenuCommands** at lines 124–134 handles the New, Close, and Quit commands in a **CASE** statement. For New and Close, it calls the proper action procedure, **DoNew** or **DoClose**. For Quit, it sets the **quitRequested** flag **TRUE**. Notice that parameter **cmdNumber** is the integer value passed by **DoCommand** representing the command position in the menu.

Procedure **DoEditMenuCommands** (137–146) has a slightly different form. It calls a Boolean function **SystemEdit**, passing parameter **cmdNumber** minus one. This allows desk accessories to recognize Edit menu commands Undo, Cut, Copy, Paste, and Clear. If **SystemEdit** returns **TRUE**, ignore the command—it's been taken care of for you. Otherwise, process the command to cut and paste information belonging to your program. Of course, in the shell, there's nothing to cut or paste. Therefore, the rest of the **IF** statement is empty (143–145).

ApShell Display Handlers (179–216)

There are only two display handlers in the shell. In a real program, this area might be the largest—it's responsible for just about everything you see during the course of a program.

The first procedure, **DrawScrollBars** (186–203), draws the window's grow box icon in the lower right corner and also the lines that form the inside borders of

vertical and horizontal scroll bars. In ApShell, there aren't any real scroll bars—only the outlines. Later examples add these and other controls. (See Chapter 5.)

DrawScrollBars takes a **WindowPtr** as its parameter indicating in which window to draw. This is necessary because the window might not be active. For active windows belonging to the program, procedures use MacExtras global variable, **whichWindow**. But that's not possible here. Consider, for example, the situation when you move a desk accessory window aside, uncovering a window underneath. In that case, the program has to redraw the scroll bar areas in the inactive window, which it does by passing the window pointer to **DrawContents**. The procedure then calls **DrawScrollBars** to redraw the newly exposed portions of the scroll bar outlines. In general, when designing procedures that might operate on non-active windows this way, pass them a window pointer. Otherwise, use the global **whichWindow** variable.

Notice the three calls to **ValidRect** in lines 200–202 and the call to **CalcControlRects** at line 199. The reason for these items will be more understandable after you read the play-by-play for the shell's Event Handlers. The statements help avoid redrawing portions of windows more than once, preventing an annoying problem that causes scroll bars to flutter briefly. I'll show you in the next section how **ValidRect** eliminates this problem.

Procedure **DrawContents** (206–216) erases the window contents and calls **DrawScrollBars**. In a real program, you'd put other commands here to draw whatever you want in the window. Notice that **DrawContents** passes the window's **portRect** to QuickDraw procedure **EraseRect**. You'll recall from the previous chapter that the **portRect** is the portion of the **GrafPort** in which QuickDraw draws. With window records, this equals the interior area of the window minus the title bar on top. Because **WindowPtr** and **GrafPtr** pointers are equivalent, you can pass the window's **portRect** to **EraseRect**, clearing the window's insides but leaving undisturbed everything else on display.

One important fact is that **DrawContents** normally draws everything that appears in the window. Remember when adding your own commands not to be concerned with *when* this might happen. In fact, **DrawContents** will be called many times—when you create a window; when you zoom it in and out; when you expose portions of it by moving other windows; and when you click it to bring it to the front. But in the Display Handler section, you do not have to observe any rules relating to these various conditions. You simply draw everything in the window and let the next section, the Event Handlers, decide when that should happen.

ApShell Event Handlers (220–339)

The shell's Event Handlers process the four main events that most programs need to recognize. Procedure **MouseDownEvents** (227–276) is first. It determines exactly where the mouse pointer was at the time you clicked the button and, from that information, calls an appropriate routine. For example, if you click inside an

inactive window, **MouseDownEvents** brings that window to the front. Or, if you choose a pull-down menu command, **MouseDownEvents** discovers that fact and responds.

The large **WITH** statement beginning at line 237 gives access to fields in Record variable **theEvent** (also from **MacExtras**) containing the data associated with this event. It's the Program Engine's responsibility to set **theEvent** record so that Event Handlers can interpret its data.

To accomplish that, the procedure calls function **FindWindow** (241) with two parameters. The first, **where** from **theEvent**, is the **Point** record with the global coordinate of the cursor hot spot—the tip of the arrow pointer or the center of a cross hair. **FindWindow** assigns to **whichWindow** (the global **MacExtras** variable) the address of the window (if any) that contains **where**, indicating the mouse click was inside that window's contents. **FindWindow** returns also a part code, locating the part number of the desktop item (the menu bar or zoom box, for example) where you clicked the mouse. The procedure assigns this value to a local **INTEGER**, **partCode**, for later use.

Inside the large **CASE** statement (243–272), part codes select one of seven possible mouse click activities. If the part code indicates a mouse click in a pull-down menu, meaning a command was chosen, the program calls **DoCommand** (246) passing the event's **where Point** record through toolbox function **MenuSelect**. This converts the record to a **LONGINT** type (containing the menu and command numbers) that **DoCommand** requires.

If the mouse click was in a system window, meaning a desk accessory probably, lines 248–249 call **SystemClick** passing the two parameters **theEvent** and **whichWindow**. **SystemClick** passes the event on to a desk accessory, for example, if you click its close button. You don't have to handle such events yourself, but you are responsible for making sure that desk accessories receive the events that belong to them.

Lines 251–254 represent the traditional way of handling mouse clicks inside program windows. As written here, if the window is not now the active one, determined by comparing **whichWindow** with **FrontWindow** (252), the program calls **SelectWindow** to bring it to the front. If it is the front window, then the program passes the mouse click location on to action procedure **DoMouseClicked**, described earlier.

With this approach, if you click inside an inactive window, you have to click a second time to accomplish something there. If you instead want to click in an inactive window, bring it to the front, and process that mouse click immediately, replace lines 251–254 with the following. Either approach is acceptable. It's your choice.

```
inContent
: BEGIN
    IF whichWindow <> FrontWindow
        THEN SelectWindow( whichWindow );
        DoMouseClicked( whichWindow )
    END;
```

Lines 256–257 drag windows to new locations when you click on the window’s title bar. **DragTheWindow**, a procedure in the MacExtras unit, takes two parameters **whichWindow** and **where**, and completely handles all window dragging details, a subject we’ll see again later. If the window is inactive, clicking the title bar also activates it unless you also hold down the Command key.

Lines 259–262 process clicks in the window’s grow box at the lower right corner, letting you resize a window by stretching its rubbery outline and then releasing the mouse button. The program does this by calling the MacExtras unit procedure **ResizeWindow** if the window is now active. If it’s not active, it calls **SelectWindow** to make it active. Because it takes two clicks to activate an inactive window and resize it, you might want to modify these lines as you did for mouse clicks in window contents.

The two remaining cases in **MouseDownEvents** handle clicks in the close and zoom boxes at either side of the window’s title bar. Lines 264–266 call **TrackGoAway**, which explodes the little go-away box, displaying the shock wave lines you see when you click it, and returning **TRUE** only if you then release the mouse button while the pointer remains inside the box. If you move the mouse and release the button, **TrackGoAway** returns **FALSE**. This gives you the chance to change your mind about closing windows. Only if you release the mouse with the arrow still in the box does the program call **DoClose** at line 266.

Lines 268–270 work similarly. In this case, two part codes **inZoomIn** and **inZoomOut** indicate the mouse was inside the window zoom box. If so, **TrackBox**, a more general form of **TrackGoAway**, returns **TRUE** if you release the button while still pointing to the box. If so, line 270 calls **ZoomInOut** (in MacExtras) to zoom the window either to full size or back to a former size.

KeyDownEvents (279–298)

Event Handler **KeyDownEvents** is much simpler than **MouseDownEvents**. It checks the event record in a **WITH** statement (288–297), pulling the character’s ASCII code out of the event message field with the statement:

```
ch := CHR( BitAnd( message, charCodeMask ) );
```

The **BitAnd** function uses global constant **charCodeMask** to extract the character code from **message**, a field in the event record. You can use a similar statement to extract the character key code by replacing **charCodeMask** with **keyCodeMask**. The character code is the key’s ASCII value; the key code is its position on the keyboard. (See Figure 4.2 for the location of these values in the **Event message** field.) Normally, the ASCII code is what you want. But there are times when you need to check the key code, sometimes known also as a *scan code*. For example, on the Macintosh Plus, there are two plus sign (+) keys that produce the same ASCII value (hex \$2B) for that symbol. To distinguish between them, use the **KeyDownEvents** procedure in Figure 4.3.

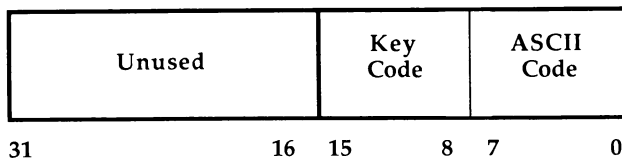
theEvent.message

Figure 4.2 Key down events return both a key code and ASCII character, stored as shown here in the low order part of the event record's **message** field.

If you have a Macintosh Plus or a numeric keypad, replace **KeyDownEvents** in a copy of ApShell with this procedure and insert **SysBeep(2)** in place of the comment, "Handle keypad plus key." Typing the keypad's plus key beeps the speaker; typing the regular plus key doesn't beep, proving that even though both keys generate the same ASCII code, they have different key codes.

After extracting the character or key code, the next step is to determine whether a modifier key was pressed at the same time. Lines 293–295 show how to check

```

PROCEDURE KeyDownEvents;

CONST
    KeyPadPlus = 70;

VAR
    keyCode : INTEGER;

BEGIN
    WITH theEvent DO
    BEGIN
        keyCode := BitAnd( message, keyCodeMask ) DIV 256;

        IF keyCode = KeyPadPlus THEN
        BEGIN
            { Handle key pad plus key }
        END ELSE
        BEGIN
            { Handle regular keypresses }
        END

        END { with }
    END; { KeyDownEvents }

```

Figure 4.3 To distinguish between keys that produce the same ASCII character code, examine their key codes as this procedure demonstrates.

Table 4.1 Event record **modifiers** constants.

BtnState	= 128;	{ Bit set if mouse button is UP }
CmdKey	= 256;	{ Bit set if command key typed }
ShiftKey	= 512;	{ Bit set if shift key typed }
AlphaLock	= 1024;	{ Bit set if caps lock key is down }
OptionKey	= 2048;	{ Bit set if option key typed }

for the Command key (**CmdKey**), calling **DoCommand** to respond to Command-Q, Command-N, and other alternatives to choosing pull-down menu commands. **KeyDownEvents** checks for the command key by logically *ANDing* the event record **modifiers** field with the constant **CmdKey** (293). If the result is not zero, then someone pressed the command key and the program calls **DoCommand**, translating the ASCII character to a menu command with the help of the toolbox **MenuKey** function. Otherwise, it calls **DoKeypress** to process normal typing.

Table 4.1 lists modifier key values that you can use to determine whether other keys were pressed. If you do this frequently, use functions such as the two in Figure 4.4 instead of directly programming **BitAnd** expressions in **KeyDownEvents**. They make programs more readable. For example, to test whether someone typed the Option and A keys, you could insert something like this at lines 293–295 in **KeyDownEvents**:

```

IF KeyedOption
  THEN DoOptionKeys ELSE
IF KeyedCommand
  THEN DoCommand( MenuKey( ch ) )
  ELSE DoKeypress( ch )

FUNCTION KeyedOption : BOOLEAN;
{ TRUE if option key held down }

  BEGIN
    KeyedOption := BitAnd( theEvent.modifiers, OptionKey ) <> 0
  END; { KeyedOption }

FUNCTION KeyedCommand : BOOLEAN;
{ TRUE if command key held down }

  BEGIN
    KeyedCommand := BitAnd( theEvent.modifiers, CmdKey ) <> 0
  END; { KeyedCommand }

```

Figure 4.4 Use functions such as these to tell whether keys like Command and Option are held down while typing.

Table 4.2 Diacritical keys.

Key	Symbol	Name	Example
`	`	grave accent	à
e	´	acute accent	é
i	^	circumflex	î
u	¨	umlaut	ü
n	~	tilde	ñ

When testing for Option keys, remember there are five predefined diacritical keys (Table 4.2). To use them, hold down the Option key, type a key in the left column of the table, let up on the Option key and type a letter. For example, holding down the Option key and typing two n's produces ñ. (See the examples in the right column.) The problem is, you cannot redefine what these key sequences do. Therefore, don't use the five keys in the table when designing your own Option key commands.

UpdateEvents (301–318)

The next Event Handler procedure, **UpdateEvents**, responds to requests for drawing a window's contents. Other procedures and programs make such requests when you uncover previously hidden windows or when you take an action that changes what windows display. For example, if you close a window, the operating system generates an update event for a window underneath. This causes **UpdateEvents** to start the sequence that redraws that window's contents.

The first job in **UpdateEvents** is to save the current **GrafPort**. The reason for this step is that update events might not be for the frontmost window. Therefore, line 310 saves a pointer to the current **GrafPort** in local variable **oldPort**. Later, line 317 restores this value just before **UpdateEvents** ends.

Lines 311–312 use type casting to convert the event record's **message** field to type **WindowPtr**, assigning this value to global variable **whichWindow**. After that, a call to procedure **SetPort** (313) changes the current **GrafPort** to the window that requires updating.

The next three steps (314–316) tell the Window Manager to begin an update for this window. The call to **BeginUpdate** (314) does these two jobs:

- Calculates the intersection of the window's visible and update regions
- Empties the update region

The visible region is that part of the window visible prior to the request for an update. The update region is the part of the window that needs redrawing (which might include portions of the window not now visible). The intersection of these

two elements exactly locates only that part of the window that is both visible and requires redrawing.

You don't have to understand or directly manipulate these regions. Just remember that when the program calls **BeginUpdate**, it limits the visible region of the **GrafPort** to the portion of the window that needs redrawing. Because this also clips, or restricts, drawing to that area, even if you redraw the entire window, you affect only the correct portion the update event requires. In other words, in your drawing routine, you can just redraw everything and let the system figure out what to actually display. This saves time, not to mention the hassle of calculating which portions of windows cover others every time you need to draw something.

Line 315 calls the drawing procedure in ApShell's Display Handler section. As programmed here, **DrawContents** merely erases the window—it doesn't draw anything. When you run the program, try moving desk accessories aside and observe what happens. Because **BeginUpdate** limits drawing to the proper areas, **DrawContents** erases and reconstructs only the necessary window parts.

Finally, **UpdateEvents** calls **EndUpdate** (316) reversing what **BeginUpdate** did. This restores the visible region of the window to indicate correctly what portions of the window are now visible. Because it complements what **BeginUpdate** does, **EndUpdate** always ends an update event after the drawing routines finish. (Note: even if you have nothing inside a window to redraw, you still must call **BeginUpdate** and **EndUpdate** to clear the event.)

ActivateEvents (321–339)

The final Event Handler procedure, **ActivateEvents**, takes care of activating some windows, bringing them to the front, and hiding others. It extracts the window pointer (329) and changes the **GrafPort** (330) to the window for this event. It does not save the current port as in **UpdateEvents** because the purpose of this procedure is to *change* that port.

Line 332 redraws any scroll bars that belong to this window, even though there aren't any such controls in ApShell. In this example, **DrawScrollBars** merely draws the outlines where scroll bars eventually go. It also redraws the grow box icon in the lower right corner.

Lines 334–336 test the event record's **modifiers** field to check whether this event applies to a window becoming active or to one that the procedure should hide. Such events usually come in pairs; the deactivate followed by the activate event. In the shell, lines 335–336 use this information to change the edit menu commands, dimming them when windows become active (335) and enabling the commands when windows become inactive. It does this because the shell doesn't allow Cut, Copy, Paste and other editing commands for its own windows. Therefore, it disables these commands. Because desk accessories might need these same commands, it enables them when windows become inactive, assuming that this might uncover a desk accessory window below.

Later, we'll see other operations that **ActivateEvents** handles. For example, it might activate or deactivate controls and scroll bars, display or remove a cursor, or highlight text for newly active windows.

Generating Update Events

Turn back to the Display Handlers section and look at lines 199–202 in procedure **DrawScrollBars**. Now that you know how programs deal with update events, you can better understand this procedure.

After drawing the grow box icon and displaying the scroll bar outlines (198), **CalcControlRects** from the MacExtras unit calculates three rectangles encompassing the horizontal scroll bar (**hBarRect**), the vertical bar (**vBarRect**), and the grow box (**gbRect**).

It does this because the procedure **DrawGrowIcon** breaks one of QuickDraw's own rules—that no routine directly draw into windows. Because an activate event often precedes an update event for a window (if a window becomes active, its contents usually need redrawing), the program inadvertently calls **DrawScrollBars** twice—once from line 332 in procedure **ActivateEvents** and then once again when **UpdateEvents** (315) calls **DrawContents** (206). Because **DrawContents** also has to redraw the scroll bars and grow icon (it does not know whether an activate event preceded it), this sometimes draws those objects twice in succession.

To see the problem, turn lines 199–202 into a comment by surrounding the statements with (* and *). Run the modified program. Open a new window and a desk accessory such as the Note Pad or Control Panel. Look closely as you move the desk accessory window aside and as you activate and deactivate the program window. You should see the grow box and scroll bar outlines shudder quickly. They do that because the program now draws them twice, once during activate events and once during updates.

To amplify the shudder, making it easier to see, add an integer variable **i** to **DrawScrollBars** and insert the following statement between lines 198 and 199.

```
FOR i := 1 TO 32000 DO;
```

ApShell solves the shuddering scroll bar problem with three **ValidRect** statements (200–202), subtracting, or *validating*, areas from the **GrafPort**'s update region. During update events, QuickDraw draws only in the area intersecting the update region (the parts that require redrawing) with the visible region (the parts you can see). By subtracting already-drawn areas from that region, QuickDraw avoids redrawing them. Therefore, when an activate event calls **DrawScrollBars**, validating the scroll bar regions prevents redrawing those same areas during subsequent updates. In general, whenever you draw directly into a window, call **ValidRect** to tell QuickDraw it does not have to update the area you just drew.

ApShell Initializations (343–393)

SetUpMenuBar (350–368) reads the menu definitions from the program's resources and inserts them into the menu bar. As you can see, creating pull-down menus is not difficult. In general, adding a new menu requires only these two steps:

- Read the menu resource
- Insert the menu into the menu bar

After performing those two steps for every menu, call **DrawMenuBar** (366) to display the program's menu. You can call this procedure again at any time. You must call it if you later make any changes to menu titles.

Menus are **MenuHandle** variables, which ApShell lets the MacExtras unit define. Most programs probably will have at least Apple, File, and Edit menus. To add others, declare new **MenuHandle** variables, create resources for them, and initialize them in **SetUpMenuBar**. Of course, you also need programming to respond to the new menu commands. Future chapters have many examples of how to do this.

Lines 356–358 load the resources for the three standard menus by calling **GetMenu**, passing the resource ID as the lone parameter. Lines 360–362 insert each of these menus into the menu bar. The 0 tells the Menu Manager to insert menus behind others. To insert a new menu to the right of another, pass its ID number instead. Usually, though, you'll simply initialize menus as the listing shows.

The program adds desk accessory (DA) names to the Apple menu by calling **AddResMenu** (364). This adds resource names—the DA titles—to the Apple menu. The string 'DRVr' (Driver) is the resource type of all desk accessories. Although there are other ways to add DA names to a menu, you should always do it as in **SetUpMenuBar**. This makes the order of menu commands identical for all programs that use the same set of DAs.

Initialize, QuitConfirmed (371–393)

ApShell calls **Initialize** (371–379) once at the start of the program. In turn, it calls **SetUpMenuBar** (376) and sets two Boolean variables **FALSE** (377–378). It could, of course, do other jobs: initialize other variables, display startup messages, or open windows. Whatever you need to do before the program starts, do it in **Initialize**.

Function **QuitConfirmed** (382–393) is the program's deinitialization routine. Its job is to ensure that the Program Engine does not end unless all conditions are right. In this case, conditions are always right and, therefore, **QuitConfirmed** simply closes any open window if variable **quitRequested** is **TRUE**.

Understand **QuitConfirmed**'s logic. If someone chooses the Quit command (**quitRequested=TRUE**), then if a window is open, **QuitConfirmed** calls

CloseProgramWindow to close it. After that, it passes the value of **quitRequested** back as **QuitConfirmed**'s function value. This makes the function mirror **quitRequested**'s state, **TRUE** or **FALSE**. In an application, **quitRequested** becomes more important. As you'll see in later examples, it needs to perform three jobs to let you:

- Return to the program
- Save changes to files
- Throw away changes to files

Any program that modifies data should follow these three steps before ending the program, avoiding potential loss of information. You've probably seen these actions in programs like MacWrite and Turbo Pascal when you choose Quit after editing a file and forgetting to save your changes.

ApShell Program Engine (397–456)

The Program Engine makes the program go. Before looking at procedure **DoSystemTasks** (404–431), it helps to understand how the Engine churns.

The first step is to call **Initialize** (436), setting up menus and doing other start-up jobs. Then a **REPEAT** loop (438–454) cycles until the program ends. Inside that loop, the Engine calls **GetNextEvent** (442), receiving events such as mouse clicks and window update requests from the operating system as they occur and passing those events on to the proper Event Handler. It does this by examining the **what** field of the event record (444) in a **CASE** statement, selecting one of five main event types. (This program ignores **AutoKey** events, generated when you hold down a key as you do in text editors to repeat characters.)

The **REPEAT** loop cycles until function **QuitConfirmed** returns **TRUE**. Notice that this simplifies the usual flag checks and double **REPEAT** loops found in other shells and examples you may have seen. You know that **QuitConfirmed** gives every opportunity to save changes, discard unwanted editing, close windows, and the like. Such jobs belong in procedures, not in the main program body.

This follows a general rule for Pascal programs: that the main body be small and easy to follow. Reading it gives you an eagle's view of the program—you see the landscape, not every detail. But don't attempt to simplify the main loop further, as many programmers do. For example, the following is a popular Program Engine:

```
BEGIN
  Initialize;
  MainLoop
END.
```

Apparently, procedure **MainLoop** handles all the **REPEAT** loop details in ApShell. While this appears to reduce the program to two steps, it's a bad design. When the program calls **MainLoop**, it creates on the stack an *activation record* which keeps track of, among other things, the location of the procedure that called it and certain variables that maintain the level of such calls throughout a program's life. You don't need to be concerned with the nature of the activation record, but you should be aware that calling **MainLoop** from the main program body this way does nothing but increase the program's nesting level by one for no good reason. (The nesting level is the depth to which a group of procedures call each other. If procedure A is on level 0 and calls B which calls C, the nesting level at that point is 2.) As ApShell shows, keeping the program loop in the main body keeps the nesting level at zero each time the Engine cycles, the best plan.

DoSystemTasks (404–431)

Procedure **DoSystemTasks** runs each time the Engine cycles. It first calls **SystemTask** (410), which gives desk accessories their fair share of time and lets them update their displays, making alarm clocks tick and debuggers debug. After that, the rest of **DoSystemTasks** performs an important task, which many programs often overlook.

If the frontmost window pointer is **NIL**, then no window is visible and the program disables the entire edit menu (416), enables the File menu's New command (417), and disables Close (418). Otherwise, if the front window is not **NIL**, then there is a window on display. But does it belong to the program? Line 422 checks by comparing the window pointer value returned by **FrontWindow** with ApShell's global **wPtr** variable. If they are not the same, then the front window must belong to a desk accessory and the program therefore enables the Edit menu commands (426). It also enables the File menu's Close command (427). That way, you can choose Close to remove desk accessories as well as clicking their close boxes.

It's helpful to remove the programming from lines 412–429 and observe the effect. Open a window and a desk accessory, then click in the window to bring the window to the front. Close the program window, leaving the desk accessory on screen, and pull down the Edit menu. As you can see, the proper commands are not activated. You cannot fix this problem by enabling the Edit menu commands in procedure **ActivateEvents** (321–339) as many people try. This is not possible because closing a window does not generate a deactivate event for it. If it did, line 336 would take care of enabling the Edit menu commands. Obviously, this doesn't happen, requiring a check for such situations at every Engine cycle.

Don't be concerned that these checks will slow the program. They won't, at least not by much. It may seem that keeping the Program Engine running efficiently is important, and it is—but only to a point. The Engine, remember, doesn't run the program statement-by-statement but, rather, on a procedural level. When it calls

a procedure to handle an event, which in turn might call another routine, the Engine completely gives up control until that procedure ends. Therefore, adding a few programming statements to highlight and dim menus has a negligible effect on the speed of those other routines.

APSHELL RESOURCES

Listing 4.2 is a resource text file, which describes the program's resources—various constants and templates that programs need to display menus, create windows, and display text such as the familiar commercial message better known as the About Program box. Many people are confused about resources. Just what are they?

I think of resources as everything that is not *consumer* data. By consumer data, I mean items such as text, database files, and spreadsheets—all of the data that people process with computers. Resources are in a different category. They contain *program* data, items that programs use during their run-time lives. Window templates, menu titles, error messages, dialog boxes, and even the program's code itself all comprise a program's resources.

Just as a program's source code differs from its compiled run-time file, its resource text differs from its binary resource file. The resource text is what you type to define a program's resources. After typing that text, you compile it with RMaker, a utility program on the Turbo disk. RMaker translates that resource text into the binary resource file, the data that the Turbo compiler then combines with your program.

Creating a Resource Text File

As you can see from Listing 4.2, this is not a Pascal program. In fact, it's not a program at all but a list of definitions. Even so, like all listings in this book, it has line numbers and colons in the left column. They are purely for reference—don't type them.

Type in Listing 4.2 with the Turbo editor and save as APSHELL.R (the R stands for Resource). Line 5 tells RMaker where to store the binary form of these resources when it compiles the text. Change line 5 to use different volume and folder names. Before typing, you might want to click off the Auto Indent option using Turbo's Options command from the Edit menu. This helps avoid accidentally typing spaces in blank lines, a situation that often confuses RMaker. If, after typing the Listing, you have trouble with RMaker, check for blank lines that aren't really blank. It's fussy about such things.

After typing the listing, run RMaker with Turbo's Transfer menu or with the

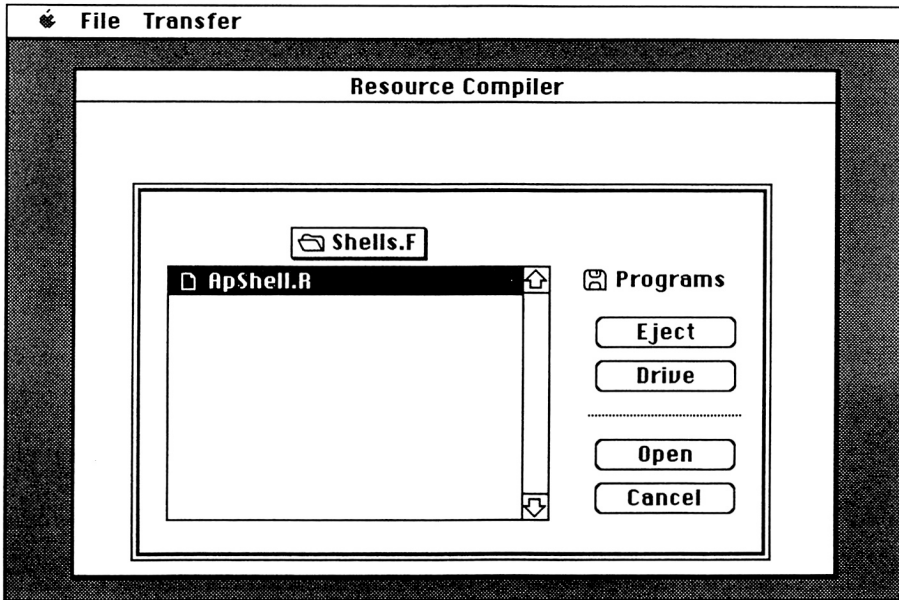


Figure 4.5 RMaker compiles a text file to produce a program's binary resources. Shown here is a copy of RMaker's main display.

File menu's Transfer command. Your display should resemble Figure 4.5. Select APSHELL.R (the file name may or may not be visible) and click the Open button to begin compiling. If your display does not resemble the figure, choose the File menu's Compile command, select APSHELL.R, and click Open.

While compiling, RMaker displays your text in the left half of its window and reports three statistics in the right half, listing the data, map, and total sizes of the finished resource file. When compiling the resource text examples in this book, if you receive any errors, check that your typing *exactly* matches the book. Unlike Pascal, where you have some freedom in inserting blank lines and changing the indentation, RMaker is unforgiving, unfriendly, and likely to reject any unplanned modifications you make. Before changing details in resource text files, then, compile them as listed here. Then make your changes. Be especially careful that blank lines do not contain any spaces, which of course you cannot see on the screen. An easy way to do this is to place the cursor to the far left on the first of a group of blank lines, cut all those lines out of the text, and then press Return to add them back. This ensures that blank lines are truly blank.

After compiling your resources, use RMaker's Transfer menu to go back to the Turbo editor. The result of compiling Listing 4.2 is a new file, APSHELL.RSRC in the volume and folder that line 5 specifies. In this file are the binary forms of

the same resources that Listing 4.2 describes. It is this binary data that Turbo combines with your program to produce a finished application.

You now have two of the three parts that make up the complete application shell. Continue to the next and final listing under the heading MacExtras Unit if you want to type in the entire program before reading the play-by-play descriptions.

Listing 4.2. APSHELL.R

```

1: *-----*
2: * ApShell.PAS resources -- Compile with RMaker      *
3: *-----*
4:
5: Programs:Shells.F:ApShell.RSRC    ;; Send output to here
6:
7:
8: *-----*
9: * About box string list                      *
10: *-----*
11:
12: TYPE STR#                                ;; String list resource
13:     ,1 (32)                            ;; ID and attribute (purgeable)
14: 6                                       ;; Number of strings that follow
15: ApShell                                ;; Program name
16: by Tom Swan                          ;; Author
17: Version 1.00                        ;; Version number
18: (C) 1987 by Swan Software            ;; Copyright notice
19: P. O. Box 206, Lititz, PA 17543      ;; Address
20: (717)-627-1911                      ;; Telephone
21:
22:
23: *-----*
24: * The Apple Info menu                      *
25: *-----*
26:
27: TYPE MENU
28:     ,1                                ;; Menu ID number to use in program
29: \14                                  ;; Bitten-apple graphics symbol
30:     About ApShell...                ;; The command as shown in menu
31:     (-                             ;; Divider line between command and DAs
32:
33:
34: *-----*
35: * The File menu                          *
36: *-----*
37:
38: TYPE MENU
39:     ,2                                ;; Menu ID number to use in program
40: File                                ;; Menu title as shown in menu bar
41:     New /N
42:     (Close
43:     (-
44:     Quit /Q
45:
46:
47: *-----*
48: * The Edit menu                          *
49: *-----*
50:

```

```

51: TYPE MENU
52:      ,3
53: Edit
54:      (Undo /Z
55:      (-
56:      (Cut /X
57:      (Copy /C
58:      (Paste /V
59:      (Clear
60:
61:
62: *-----*
63: * Window template                                *
64: *-----*
65:
66: TYPE WIND
67:      ,1 (32)                ;; ID number and attribute (purgeable)
68: Untitled                    ;; Window title
69: 46 7 328 502                ;; top, left, bottom, right coordinates
70: Visible GoAway              ;; Visible window with close button
71: 8                            ;; Standard doc window with grow & zoom boxes
72: 0                            ;; Window reference (none)
73:
74:
75:
76: * END

```

ApShell Resource Play-by-Play

Resource text always begins with the file name to which you want RMaker to compile the resources to disk. Line 5 places ApShell's resources on disk volume Programs in folder Shells.F in the file APSHELL.RSRC.

Following this required line, you can insert whatever resources you like. In my own designs, I start with six strings, identifying the program name, version, author, copyright notice, and other information. When the program displays its "About program . . ." box, it displays these strings.

Lines 12–13 show how all resource definitions begin. First comes the word **TYPE** followed by a 4-character string that identifies the resource variety. **STR#** means *string list*. The *Guide* and *Inside Macintosh* list other resource types. (We'll see many of them in future examples.) After specifying the resource type, line 13 lists its ID number, in this case a 1 preceded by a comma. The indentation makes ID numbers stand out on the page but it's not required. These numbers are the same as the numbers programs use to locate resource definitions. They form a link between the program and its resources.

All such ID numbers must be different for any one resource type. For example, look at lines 27–28. This menu resource ID number (1) is the same as the string list's number in line 13. Because they are different resource types, they may have the same numbers. But all Menu resources—as well as other categories—must have different ID numbers within that category. You cannot have two menus or two windows with the same ID.

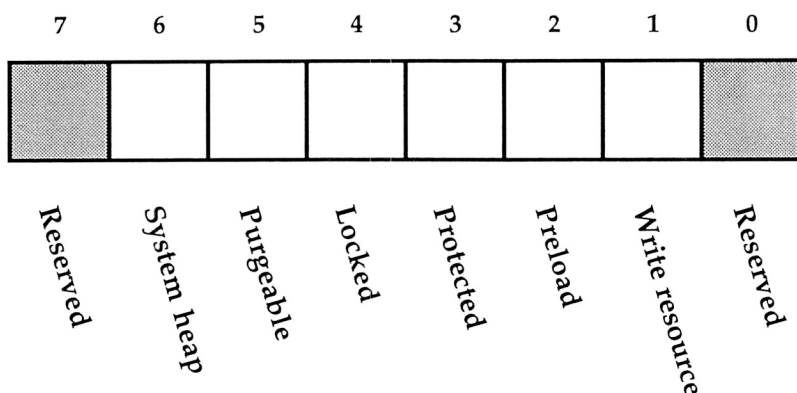


Figure 4.6 Resource attribute bits have these meanings. The shaded cells are not used.

After the ID is an optional decimal number in parentheses, the *resource attribute*. Figure 4.6 describes what the bits in these numbers mean. The shaded cells are reserved—don’t use them. Setting a bit to 1 activates that option. For example, binary value 00100100, or \$24 hex (36 decimal) represents the options “Purgeable, Preload.” Although examples in this book do not use all possible attribute combinations, the following are brief descriptions of what the various settings offer. The numbers in parentheses after the labels are the values to use with the ID number in the resource text. Add these values together to combine attributes. (See *Inside Macintosh* for more information.)

System heap (64). This loads the resource onto the system heap (1) or onto the application heap (0), the normal and usually recommended setting.

Purgeable (32). This allows the memory manager to throw out memory that resources occupy (1) anytime after loading the resource from disk. To prevent this from happening, set bit 5 to 0. Use this setting only for resources that the system copies before using, for example, strings. Others that might be used directly (menu resources and some others) must never be purgeable.

Locked (16). Set bit 4 to lock the resource memory on the heap (1), preventing the memory manager from moving it to make room for other objects. Use this option only in rare circumstances. Normally, reset this bit (0) to let the memory manager make efficient use of memory. Using this option overrides the *Purgeable* setting because memory cannot be locked on the heap and purgeable at the same time. (See Chapter 5 for more information about the heap and Memory Manager.)

Protected (8). This disallows (1) changes to the resource or enables (0) programs to make changes. Normally set it to zero. Because programs can themselves change resource attributes, resetting the protected bit if they want, this option cannot fully protect your resources from modification.

Preload (4). This immediately reads a resource into memory when opening the resource file (1) or leaves the resource on disk only until the program requires it (0). In Turbo Pascal, the resource file is open when the program runs. Normally, this bit is zero except for error dialogs (see Chapter 6) that display messages such as “Disk error,” which you might not be able to read after such an error occurs. In that case, you can preload the message into memory by setting attribute bit 2.

Write resource (2). This tells the system to write changed resources to disk (1), usually when the program ends. Never directly set this bit—the toolbox resource manager uses it in programs that change resources on the fly.

Notice that line 13 in Listing 4.2 specifies the string list to be *purgeable*. (32 is \$20 in hex, the byte value with Figure 4.6’s Purgeable bit set to one.) This means that after reading and displaying the strings, the memory space they occupy is available for other uses. The next time the Resource Manager attempts to read this same string list, it might find them gone and will have to reload them from disk. In the meantime, though, if no other process used the memory the strings occupy, the memory manager reuses the in-memory copy—it doesn’t automatically reload from disk.

Lines 23–59 define menu titles and commands. These resources must *not* be purgeable because the Menu Manager refers directly to their definitions in memory. Lines 27–28 define the Apple menu, giving it ID number 1. Line 29’s cryptic \14 stands for the bitten-apple symbol, the Macintosh’s menu bar trademark in the upper left corner. In other menus (see lines 40 and 53), you’ll usually type the full menu title at this position.

After these three lines come the menu commands. In this first menu, there is only one, About ApShell. The indentation makes the resource text more readable. But if you prefer, type everything flush against the left border. Line 31 is a divider line, which, although it takes up one menu position, is permanently inactive. The dash (–) tells the Menu Manager to draw a line at this position. The preceding left parenthesis tells it to disable this item. If you don’t include the parenthesis, the divider line becomes an active menu command. Try this on a copy of AP-SHELL.PAS and APSHELL.R. When you run the program, select the divider line from the Apple menu. Obviously, this is a mistake.

The File and Edit menus follow a similar design. Notice that a few commands such as Close and Cut have left parentheses, disabling these commands when the program starts. Later, the program enables some commands at appropriate times and disables others. The slashed character endings to some commands (/N and /Q for example) specify equivalent Command keys. The Menu Manager ignores case when you type these keys, but it’s traditional to enter them in uppercase.

You don't have to do anything special to enable command keys. ApShell automatically handles them for you. If you want to type Command-P to print, for example, just add /P after that menu command and ApShell does the rest. Be careful not to duplicate Command keys in different menus, though. Unfortunately, RMaker is not smart enough to notice this error.

The final resource definition is a window template (66–72). It has type **WIND**, ID number 1, and is purgeable. Remember that the resource is only a template. The Window Manager creates the actual window record in memory from this information—it doesn't need it afterwards. If you didn't make it purgeable, the template would remain in memory even though the program might never need it again, causing no harm but wasting space.

At line 68 is the window title, which programs can easily change as you'll see in future examples. Line 69 declares the top, left, bottom, and right coordinates of where you want the window contents to appear. Remember that these coordinates do not include the window's borders or its title bar, features the Window Manager adds when it creates and displays the window. If you want a window's content area exactly 150 pixels wide, for example, just make right–left equal to 150. Similarly, if you want a 200-pixel tall window, make sure bottom–top equals 200. Remembering this simple fact helps define windows exactly the size you want.

Line 70 of the window resource definition declares whether the window should immediately become visible when the program creates it and whether it should have a close or GoAway box in the upper left corner. Alternatively, you could specify Invisible and NoGoAway. You normally make windows invisible if the program relocates and resizes them immediately after reading the resource definition. To display the window, the program would then call toolbox procedure **ShowWindow**.

Line 71 declares the window's *definition ID*, a value that the Window Manager uses to load various internal routines that take care of drawing window borders, title bars, and other jobs. (Figure 5.7 in the next chapter lists other definition IDs you can use to display a variety of window styles.) The value 8 here specifies a standard document window with a resize box (and scroll bar outlines) and a zoom box in the upper right corner. If you don't want a zoom box, use zero instead. In either case, it won't appear on 64K ROM Macintoshes.

The final line of the window resource definition (72) is a zero, the window reference. Window records contain a 32-bit field into which you can store whatever you like. Programs often use it to keep pointers to items (usually text) associated with the window. (Chapter 5 explains how to do this.) Rather than keep track of associated data by other means, which you certainly can do, this lets windows point to their own data, simplifying the program. You can store any other 32-bit value here—it doesn't have to be a pointer variable.

The last line, 76, is not required. I add it here so you know this is the end of the resource text. Notice that it begins with an asterisk (*), causing RMaker to ignore everything afterward. Other comments begin with double semicolons (;). Be careful if you leave the comments out, by the way. Some resource types such as

PAT and STR are three characters instead of four, the required length for all type names. In that case, a line such as this:

```
TYPE STR    ;;this is a comment
```

lets RMaker recognize STR as a four-character ID, because at least one blank space follows the name (between STR and ;;). But, if you remove the comment and write:

```
TYPE STR
```

RMaker refuses to compile the resource text unless you explicitly type a blank space after STR. Such idiosyncrasies have given resources a bad name and lead many programmers not to use them. That's unfortunate. Resources are vital to writing good Macintosh software. They isolate strings for translating into other human languages and they help the Memory Manager purge unneeded data from memory, making room for other routines. They define templates for windows and dialogs, reducing the amount of programming needed to create such items. It is true, though, that RMaker is cranky, ornery, and difficult to use. If you receive errors, check every character and be sure your file exactly matches the listings here.

MACEXTRAS UNIT

Listing 4.3 collects various constants, types, variables, procedures, and functions that are unlikely to change from one program to another. By putting these items into a unit—instead of putting them into APSHELL.PAS—you avoid recompiling these unchanging parts every time you compile a new program. To add the common elements to programs, insert MacExtras into the **USES** clause as you do for other units such as QuickDraw and ToolIntf.

Type in Listing 4.3 and save as MACEXTRAS.PAS. Change line 1 to use different volume and folder names. Compile to disk. If you instead compile a unit to memory, you can still compile programs that use the unit's features. But, unless you compile to disk, you have to repeat that step every time you start Turbo to compile and run programs. This defeats the purpose of using units—to save time by precompiling common routines. For that reason, always compile units to disk code files. Also, be aware that running units has no effect. They are not stand-alone programs but more like libraries that contain parts and pieces other programs can use.

After compiling MacExtras, you're ready to compile and run ApShell. If you're typing these listings without reading the descriptions, return to ApShell play-by-play for information about using and modifying the shell.

Listing 4.3. MACEXTRAS.PAS

```

1: {$O Programs:Units.F: }      { Send compiled code to here }
2: {$U-}                        { Turn off standard library units }
3:
4:
5: UNIT MacExtras( 128 );
6:
7: (*
8:
9:  * PURPOSE : Miscellaneous routines
10:  * SYSTEM  : Macintosh / Turbo Pascal
11:  * AUTHOR   : Tom Swan
12:
13:  *)
14:
15:
16: INTERFACE                      { Items visible to a host program }
17:
18:
19:     USES
20:
21:         Memtypes, QuickDraw, OSIntf, ToolIntf, PackIntf;
22:
23:
24:     CONST
25:
26:         AppleID      = 1;      { Apple menu resource ID and commands }
27:         AboutCmd      = 1;
28:
29:         EditID       = 3;      { Edit menu resource ID and commands }
30:         UndoCmd       = 1;
31:         {-----}
32:         CutCmd        = 3;
33:         CopyCmd       = 4;
34:         PasteCmd      = 5;
35:         ClearCmd      = 6;
36:
37:         ScBarWidth    = 15;    { Width of a scroll bar }
38:         MenuBarWidth  = 18;    { Width of a window title bar }
39:         MaxMenuCmds   = 31;    { Maximum commands in a pull-down menu }
40:
41:
42:     TYPE
43:
44:         MenuCmdSet = SET OF 1 .. MaxMenuCmds;
45:
46:
47:     VAR
48:
49:         appleMenu      : MenuHandle;      { Handles to menus }
50:         fileMenu        : MenuHandle;
51:         editMenu        : MenuHandle;
52:
53:         theEvent        : EventRecord;    { Events from operating system }
54:         whichWindow     : WindowPtr;      { Window applying to event }
55:
56:
57: FUNCTION InRange( n, min, max : INTEGER ) : BOOLEAN;
58:
59: PROCEDURE Pause;
60:
61: PROCEDURE EnableMenu( mh : MenuHandle; commands : MenuCmdSet );
62:
63: PROCEDURE DisableMenu( mh : MenuHandle; commands : MenuCmdSet );
64:

```

```

65: PROCEDURE FixEditMenu( enableCommands : BOOLEAN );
66:
67: PROCEDURE DragTheWindow( whichWindow : WindowPtr; startPoint : Point );
68:
69: PROCEDURE ResizeWindow( whichWindow : WindowPtr; startPoint : Point );
70:
71: PROCEDURE ZoomInOut( whichWindow : WindowPtr; partCode : INTEGER );
72:
73: PROCEDURE CloseDAWindow;
74:
75: PROCEDURE GetPortSize( VAR width, height : INTEGER );
76:
77: PROCEDURE CalcControlRects( whichWindow : WindowPtr;
78:     VAR hBarRect, vBarRect, gbRect : Rect );
79:
80: FUNCTION TextHeight( wPtr : WindowPtr ) : INTEGER;
81:
82: PROCEDURE CenterString( h, v, w : INTEGER; s : Str255 );
83:
84: PROCEDURE DisplayAboutBox;
85:
86: PROCEDURE DoAppleMenuCommands( cmdNumber : INTEGER );
87:
88:
89:
90: IMPLEMENTATION      { Items not visible to a host program }
91:
92:
93: FUNCTION InRange;
94:
95: { Returns TRUE if min <= n <= max }
96:
97:     BEGIN
98:         InRange := ( min <= n ) AND ( n <= max )
99:     END; { InRange }
100:
101:
102: PROCEDURE Pause;
103:
104: { Wait for mouse button click. Also clear keyboard events. }
105:
106:     BEGIN
107:
108:         WHILE Button DO SystemTask;
109:         WHILE NOT Button DO SystemTask;
110:         FlushEvents( keyDownMask + autoKeyMask, 0 )
111:
112:     END; { Pause }
113:
114:
115: PROCEDURE EnableMenu;
116:
117: { Enable these commands in this menu }
118:
119:     VAR
120:
121:         theCommand : 1 .. MaxMenuCmds;
122:
123:     BEGIN
124:         FOR theCommand := 1 TO MaxMenuCmds DO
125:             IF theCommand IN commands
126:                 THEN EnableItem( mh, theCommand )
127:         END; { EnableMenu }
128:
129:
130: PROCEDURE DisableMenu;
131:

```

(continued)

```

132: { Disable these commands in this menu }
133:
134:   VAR
135:
136:     theCommand : 1 .. MaxMenuCmds;
137:
138:   BEGIN
139:     FOR theCommand := 1 TO MaxMenuCmds DO
140:       IF theCommand IN commands
141:         THEN DisableItem( mh, theCommand )
142:       END; { EnableMenu }
143:
144:
145: PROCEDURE FixEditMenu;
146:
147: { Enable commands in Edit menu if param is TRUE, else disable them }
148:
149:   VAR
150:
151:     editSet : MenuCmdSet;
152:
153:   BEGIN
154:     editSet := [ UndoCmd, CutCmd, CopyCmd, PasteCmd, ClearCmd ];
155:     IF enableCommands
156:       THEN EnableMenu( editMenu, editSet )
157:       ELSE DisableMenu( editMenu, editSet )
158:     END; { FixEditMenu }
159:
160:
161: PROCEDURE DragTheWindow;
162:
163: { Respond to mouse click in window's drag bar }
164:
165:   VAR
166:
167:     limitRect : Rect;           { Limits window location }
168:
169:   BEGIN
170:     WITH screenBits.bounds DO
171:       SetRect( limitRect, left + 4, top + 24, right - 4, bottom - 4 );
172:       DragWindow( whichWindow, startPoint, limitRect )
173:     END; { DragTheWindow }
174:
175:
176: PROCEDURE ResizeWindow;
177:
178: { Respond to clicking and dragging a window's grow box }
179:
180:   VAR
181:
182:     size      : LONGINT;        { Window's new size }
183:     width, height : INTEGER;    { Extracted from size }
184:     limitRect  : Rect;         { Limits min/max window size }
185:
186:   BEGIN
187:     WITH screenBits.bounds DO
188:       SetRect( limitRect, 100, 75, right, bottom - 24 );
189:       size := GrowWindow( whichWindow, startPoint, limitRect );
190:       IF size <> 0 THEN
191:         WITH whichWindow^ DO
192:           BEGIN
193:             EraseRect( portRect );
194:             width := LoWord( size );
195:             height := HiWord( size );
196:             SizeWindow( whichWindow, width, height, TRUE );
197:             InvalRect( portRect )
198:           END { if }
199:         END; { ResizeWindow }

```

```

200:
201:
202: PROCEDURE ZoomInOut;
203:
204: { Zoom window in (partCode=inZoomIn) or out (partCode=inZoomOut).
205:   Called only for windows with zoom boxes on Macs with 128K ROMS }
206:
207:   VAR
208:
209:       oldPort : GrafPtr;      { For saving/restoring port }
210:
211:   BEGIN
212:       GetPort( oldPort );      { Save current port }
213:       SetPort( whichWindow );
214:       EraseRect( whichWindow^.portRect );
215:       ZoomWindow( whichWindow, partCode, TRUE );
216:       SetPort( oldPort )      { Restore original port }
217:   END; { ZoomInOut }
218:
219:
220: PROCEDURE CloseDAWindow;
221:
222: { Close a desk accessory window usually in response to a menu close
223:   command when a desk accessory is the active window. It's up to you
224:   to be sure a DA is the front window before calling this procedure. }
225:
226:   VAR
227:
228:       DANumber : INTEGER;      { Desk accessory reference number }
229:       DAWindow : WindowPeek;   { Pointer to DA window record }
230:
231:   BEGIN
232:
233:       DAWindow := WindowPeek( FrontWindow );
234:       DANumber := DAWindow^.windowKind;
235:       CloseDeskAcc( DANumber )
236:
237:   END; { CloseDAWindow }
238:
239:
240: PROCEDURE GetPortSize;
241:
242: { Return width and height of current grafPort in pixels. If this is
243:   a window, then the width equals the number of pixels between the right
244:   and left borders and the height excludes the window's title bar. }
245:
246:   BEGIN
247:       WITH thePort^.portRect DO
248:       BEGIN
249:           width := right - left;
250:           height := bottom - top
251:       END { with }
252:   END; { GetPortSize }
253:
254:
255: PROCEDURE CalcControlRects;
256:
257: { Calculate horizontal, vertical, and grow box rectangles for
258:   this window. }
259:
260:   BEGIN
261:       WITH whichWindow^.portRect DO
262:       BEGIN
263:
264:           gbRect.top := bottom - ScBarWidth;
265:           gbRect.left := right - ScBarWidth;
266:           gbRect.bottom := bottom;
267:           gbRect.right := right;

```

(continued)


```

268:
269:         hBarRect := gbRect;
270:         hBarRect.left := left;
271:         hBarRect.right := gbRect.left;
272:
273:         vBarRect := gbRect;
274:         vBarRect.top := top;
275:         vBarRect.bottom := gbRect.top
276:
277:         END { with }
278:     END; { CalcControlRects }
279:
280:
281: FUNCTION TextHeight;
282:
283: { Returns the height in pixels of the text font for this window }
284:
285:     VAR
286:
287:         fInfo : FontInfo;      { Holds information about current font }
288:         oldPort : GrafPtr;     { For saving and restoring current port }
289:
290:     BEGIN
291:         GetPort( oldPort );
292:         SetPort( wPtr );
293:         GetFontInfo( fInfo );
294:         WITH fInfo DO
295:             TextHeight := ascent + descent + leading;
296:             SetPort( oldPort )
297:         END; { TextHeight }
298:
299:
300: PROCEDURE CenterString;
301:
302: { Center a string in a window or in a portion of a window }
303:
304:     BEGIN
305:         w := w - StringWidth( s );
306:         IF w < 0
307:             THEN w := 0;
308:         MoveTo( h + ( w DIV 2 ), v );
309:         DrawString( s )
310:     END; { CenterString }
311:
312:
313: PROCEDURE DisplayAboutBox;
314:
315: { Display information about program and author, usually in
316: response to selecting the About program... command from the
317: Apple menu. Requires 6-string STR# resource with ID = 1
318: containing strings to display in window as follows:
319:
320:     STR# 1   = Program name           ex.   MacProgram
321:     STR# 2   = Author                  ex.   by Dee Bugger
322:     STR# 3   = Version                  ex.   Version 1.00a
323:     STR# 4   = Copyright                ex.   1987 by NoWare Inc.
324:     STR# 5   = Address                  ex.   POB 9, NoWareLand, NW, 00000
325:     STR# 6   = Phone number            ex.   212-555-1212
326: }
327:
328:     CONST
329:
330:         StrListID = 1;      { Resource ID of STR# (string list) resource }
331:
332:     VAR
333:

```

```

334:      oldPort  : GrafPtr;          { For saving / restoring port }
335:      wp       : WindowPtr;        { Pointer to following wRec }
336:      wRec     : WindowRecord;      { Window details }
337:      wr       : Rect;              { Enclosing rectangle }
338:      i        : INTEGER;           { Misc. FOR loop control }
339:
340:      messages : ARRAY[ 1 .. 6 ] OF Str255; { Resource strings }
341:
342: BEGIN
343:
344:     FOR i := 1 TO 6 DO
345:         GetIndString( messages[ i ], strListID, i );
346:
347:         wr := screenBits.bounds;
348:         InsetRect( wr, 100, 75 );
349:         wp := NewWindow(
350:             @wRec, wr, '', TRUE, altDBBoxProc, POINTER(-1), FALSE, 0 );
351:
352:         IF wp <> NIL THEN WITH wp^.portRect DO
353:             BEGIN
354:
355:                 GetPort( oldPort ); { Save current port }
356:                 SetPort( wp );      { Make sure our window is the port }
357:
358:                 TextFont( systemFont ); { Display title }
359:                 TextSize( 12 );
360:                 CenterString( 0, 30, right, messages[ 1 ] );
361:
362:                 TextFont( geneva ); { Display other info }
363:                 TextSize( 9 );
364:                 CenterString( 0, 60, right, messages[ 2 ] );
365:                 CenterString( 0, 90, right, messages[ 3 ] );
366:                 CenterString( 0, bottom-60, right, messages[ 4 ] );
367:                 CenterString( 0, bottom-40, right, messages[ 5 ] );
368:                 CenterString( 0, bottom-20, right, messages[ 6 ] );
369:
370:                 Pause; { Wait for mouse click }
371:                 CloseWindow( wp ); { Erase "About box" window }
372:                 SetPort( oldPort ) { Restore old port }
373:
374:             END { with }
375:
376:         END; { DisplayAboutBox }
377:
378:
379: PROCEDURE DoAppleMenuCommands;
380:
381: { Execute command in the Apple menu }
382:
383: VAR
384:
385:     daName  : Str255; { Desk accessory name }
386:     result  : INTEGER; { Value ignored }
387:
388: BEGIN
389:     IF cmdNumber = AboutCmd
390:     THEN
391:         DisplayAboutBox
392:     ELSE
393:         BEGIN { Open a desk accessory }
394:
395:             IF FrontWindow = NIL
396:             THEN FixEditMenu( TRUE ); { Enable edit commands }
397:
398:             GetItem( appleMenu, cmdNumber, daName );
399:             result := OpenDeskAcc( daName )
400:
401:         END { else }
402:     END; { DoAppleMenuCommands }

```

(continued)

```

403:
404:
405: BEGIN
406:
407: { Initialize toolbox managers. Order is critical. }
408:
409:   InitGraf( @thePort );           { Initialize Quickdraw }
410:   InitFonts;                      { Initialize Font manager }
411:   InitWindows;                   { Initialize Window manager }
412:   InitMenus;                     { Initialize Menu manager }
413:   TEInit;                        { Initialize TextEdit routines }
414:   InitDialogs( NIL );            { Initialize Dialog manager }
415:   InitCursor;                   { Change cursor to a visible arrow }
416:   FlushEvents( everyEvent, 0 ) { Ignore any pending events }
417:
418: END. { MacExtras unit }

```

MacExtras Play-by-Play

MacExtras defines two standard menus, unlikely to change from one program to another. These are the Apple (26–27) and Edit (29–35) menus, with the usual commands Macintosh owners know by heart ten minutes after unpacking their computers. Notice that AppleID is 1 and EditID is 3. These values relate to the resource menu IDs as Listing 4.2 shows (lines 28 and 52). Normally, programs also have a File menu with resource ID 2. But, this is not a requirement and, therefore, MacExtras does not define its resource ID constant.

Three other values define the width of vertical and horizontal scroll bars (**ScBar-Width**), the width of a window title bar (**MenuBarWidth**), and the maximum number of menu commands that any one menu can have (**MaxMenuCmds**). If you design programs with this many commands, newer Macintoshes with 128K ROMS automatically scroll them up and down inside the pop-up menu window, a feature of dubious value because of the time it takes to select items at the bottom.

The lone data type, **MenuCmdSet** (44), defines a set of menu numbers from one to the value of **MaxMenuCmds**. Programs can use these sets to enable and disable groups of menu commands (see procedures **EnableMenu** and **DisableMenu**).

The unit defines three menu handles, **appleMenu**, **fileMenu**, and **editMenu** (49–51). Use these variables to operate on menus, add check marks, disable commands, and so on. In ApShell, procedure **SetUpMenuBar** (350–368) initializes these variables to locate the program's menu resources.

Two extremely important variables appear at lines 53–54. Variable **theEvent** is of type **EventRecord**, which fully describes events as they occur, storing such items as the mouse location, a handle to a window that requires updating, and characters typed on the keyboard. You saw examples of such event descriptions in ApShell's Event Handlers (220–339).

Line 54 declares a single **WindowPtr** variable, **whichWindow**, another important item. This pointer addresses the window that applies to the most recent mouse down (**MouseDown**), update (**UpdateEvt**), and activate (**ActivateEvt**) events. Pro-

gram Action and Display Handler routines use **whichWindow** to operate in the proper window and to avoid operating in those that don't belong to the program.

Even though MacExtras defines a single global **whichWindow** variable, this does not mean programs can have only one window. As programmed here, ApShell limits you to a single window, a restriction that's easy to change as future examples show. The **whichWindow** global variable addresses one window from *all* the windows the program uses. Use it to know which window applies to an event—don't use it as your program's fixed pointer to a window it opens. (See procedure **DoNew** in ApShell (77–93). It assigns a pointer to variable **wPtr**, *not* to **whichWindow**.)

The following notes describe each of the procedures in MacExtras. For reference, the procedure declarations are repeated along with their parameters.

```
FUNCTION InRange( n, min, max : INTEGER ) :
  BOOLEAN;
```

InRange returns **TRUE** if value **n** is inclusively between **min** and **max**. Mathematically speaking, **InRange** is **TRUE** if the expression $\text{min} \leq n \leq \text{max}$ is also **TRUE**. Use the function to avoid **IF** statements such as this:

```
IF ( i > 0 ) AND ( i <= 100 )
  THEN DoSomething
  ELSE DisplayError;
```

Instead, use **InRange** to make the program more readable.

```
IF InRange( i, 1, 100 )
  THEN DoSomething
  ELSE DisplayError;
```

```
PROCEDURE Pause;
```

Pause waits for you to click the mouse button. It's useful to insert pauses in routines such as **DisplayAboutBox** (313–376) that traditionally end when you click the mouse. You can use it also as a debugging device. For example, insert **Pause** at places where you want to slow down the program. For a test, insert the statement **Pause;** (with a semicolon) between lines 197 and 198 in a copy of ApShell and turn lines 199–202 into a comment—the same experiment you typed earlier to see the scroll box shudder problem. Run the new program and open a window. Notice that the scroll bar outlines now display twice in succession when you click the mouse button. Remember to use **Pause** in other situations when you suspect a sequence of events occurs in an unexpected order.

If you examine **Pause**'s programming, you might think lines 108 and 109 strange. The first **WHILE** statement waits for you to *release* the mouse button in case you insert a **Pause** in the program where the mouse might already be down (after click-

ing the close box, for example). The second **WHILE** waits for you to press the mouse button. If the mouse button is up, then the first **WHILE** loop has no effect. The call to **SystemTask** in each case gives desk accessories their time share while **Pause** waits for you to press or release the button. This lets the alarm clock and other accessories update their displays and follows the general rule that you should call **SystemTask** inside of loops that might take longer than 1/60-second to end.

The final statement in **Pause** flushes all keyboard events from the event queue. The reason for this step is more obvious if you turn it into a comment, recompile the unit, and then run ApShell. Run the program and open a window, then select the About ApShell command from the Apple menu. Type Command-C, click the mouse, and the window closes! What happened is that keypresses make it into the event queue but aren't noticed by the Program Engine until **Pause** ends. This can produce surprises as the experiment proves.

```
PROCEDURE EnableMenu( mh : MenuHandle;
  commands : MenuCmdSet );
```

Call **EnableMenu** to activate a set of menu commands. Pass in **mh** a menu handle such as **AppleMenu** or **EditMenu**. Pass in **commands** the set of command numbers that you want to enable. These numbers represent the positions of the commands in the menu, with the first command equal to one. Remember that divider lines count as menu commands even though you cannot normally choose them. For example, to enable the Undo and Paste commands in the Edit menu, you could write:

```
EnableMenu( editMenu, [ UndoCmd, PasteCmd ] );
```

This does not affect any other menu commands nor commands already enabled. The procedure differs from the usual method of enabling commands one by one because it lets you specify a set of commands, either one, two, or all the commands in the menu with a single procedure call. Another way to do this is to call the toolbox **EnableItem** procedure as follows:

```
EnableItem( editMenu, UndoCmd );
EnableItem( editMenu, PasteCmd );
```

Using **EnableMenu** avoids long sequences of such statements often found in Macintosh programs. One trick you can use is to enable entire menus with this statement:

```
EnableMenu( editMenu, [ 1 .. MaxMenuCmds ] );
```

This works because **EditItem**, which **EnableMenu** calls, ignores commands that don't exist in menus. But don't do this in menus that have divider lines or you'll enable them, too!

```
PROCEDURE DisableMenu( mh : MenuHandle ;
  commands : MenuCmdSet ) ;
```

DisableMenu is similar to **EnableMenu** except that it dims a set of menu commands. As with **EnableMenu**, to disable all commands in a menu, use the following statement, which works even if the menu contains divider lines as these are disabled anyway:

```
DisableMenu( editMenu, [ 1 .. MaxMenuCmd ] ) ;
```

```
PROCEDURE FixEditMenu( enableCommands :
  BOOLEAN ) ;
```

This procedure enables the standard editing commands (see lines 30–35) if **enableCommands** is **TRUE** or disables them if it's **FALSE**. This is handy especially for programs that don't enable their own editing commands but follow ApShell's plan of enabling them for desk accessories.

```
PROCEDURE DragTheWindow( whichWindow :
  WindowPtr ; startPoint : Point ) ;
```

ApShell calls **DragTheWindow** in response to mouse clicks inside the window's title bar (but outside any close or zoom box). As long as you hold down the mouse button, the procedure draws an outline of the window and lets you move it to a new position.

Actually, toolbox procedure **DragWindow** handles most of the action. This procedure merely sets rectangle **limitRect** (167) to a standard size four pixels in from the left, right, and bottom borders, and 24 pixels down from the top. This tells **DragWindow** to limit dragging to an area that prevents moving windows so far they disappear off screen—obviously a bad situation. If you always call **DragTheWindow** to drag windows, this will never happen.

Notice that the procedure uses the toolbox global variable **screenBits.bounds** (a **Rect** field) to get the display's current boundaries (170). This field always indicates the limits of the Macintosh display and, presumably, of past and future models with different screen sizes.

```
PROCEDURE ResizeWindow( whichWindow :
  WindowPtr ; startPoint : Point ) ;
```

When you click and drag a window's resize box in the lower right corner, this procedure handles the details of drawing a window's outline and resizing the window borders when you release the button. ApShell calls it automatically for all windows with resize buttons (also called grow boxes). You never have to call it yourself.

As in **DragTheWindow**, **ResizeWindow** relies on a toolbox procedure to do most

of its work. Likewise, it begins by setting a rectangle, **limitRect** (187–188), to an area that in this case limits the maximum and minimum sizes you can stretch and shrink window boundaries. Although **limitRect** is a standard rectangle variable, toolbox procedure **GrowWindow** (189) uses its values differently than usual. In this case the fields have these meanings:

- left—minimum width
- top—minimum height
- right—maximum width
- bottom—maximum height

As programmed here, windows can be no smaller than 75 by 100 pixels and no larger than the full screen width minus the 24-pixel menu bar. This means you cannot make windows larger than the visible screen although there is no danger if that should happen.

Most of the action occurs at line 189 with the call to **GrowWindow**, which requires a handle to the window (**whichWindow**), the location of the mouse (**startPoint**), and the limiting rectangle (**limitRect**) as parameters. The function returns a **LONGINT** variable that tells if the window size changed and, if so, what the new size is. It retains control as long as you hold down the mouse button.

After you release the button, **ResizeWindow** checks **size** (190). If zero, then the window size did not change and nothing else needs to be done. If not zero, then the statements at lines 193–197 set into motion the actions that eventually redraw the window in its new size. First, the procedure erases the entire window contents, passing the **portRect** field from the window's **GrafPort** to procedure **EraseRect** (193). This step is optional, but it smooths the resizing steps by clearing everything out of the old window before its size changes. If you want to experiment with the effect this statement has, turn it into a comment and insert **Pause**; between lines 192 and 193 in MacExtras and between lines 210 and 211 in ApShell. Open a window and change its size. Move the mouse pointer aside and click twice to see the series of actions that grow and shrink windows. Notice that when you make the window larger, the old scroll bar outlines clutter up the window while it grows. In high speed, this looks choppy. Now, reenact **EraseRect** in MacExtras (193) but keep the **Pause**; statements in place. This time, when you enlarge a window, the program erases the scroll bar outlines, making a smoother animation.

```
PROCEDURE ZoomInOut( whichWindow : WindowPtr;
  partCode : INTEGER );
```

ApShell calls **ZoomInOut** when you click a window's zoom box in the upper right corner. This zooms windows to full screen or back again to original size. Never call this procedure on Macintoshes with 64K ROM—the routines it calls exist only

```

FUNCTION Has128KROM : BOOLEAN;

{ TRUE if version number >= 117 }

VAR

    rom, machine    : INTEGER;      { System identifiers }

BEGIN
    Environs( rom, machine );      { Get system environment info }
    Has128KROM := ( rom >= 117 )
END; { Has128KROM }

```

Figure 4.7 Before calling tool **ZoomInOut** in unit MacExtras, use this function to determine whether the computer has 128K ROMS, a zooming prerequisite.

in 128K ROMS (and, with any luck, in future ROM versions too). Normally, you'll never call this procedure directly although you certainly can if you want. For example, you might add a Zoom-Window command similar to Turbo's and call **ZoomInOut** in response.

If you do that, use the function in Figure 4.7 to check whether the computer has the new 128K ROMS before calling **ZoomInOut**. (You might want to add this function to MacExtras if you plan to use it often.) It works by calling **Environs**, which returns the system environment—namely the ROM version number and machine ID. We'll see this command again in the next chapter in a program that lists various information about your system. But here, we just want to check the ROM version. If it's at least 117, then the computer has the 128K ROMS and can zoom windows in and out. Otherwise, it has older ROMS and your program must not call **ZoomInOut**.

ZoomInOut saves and restores the current **GrafPort** (212–213, 216), a possibly unnecessary step if you never explicitly call **ZoomInOut** but let ApShell call it only when someone clicks the zoom box. Because the zoom box never appears unless the window is active, the current **GrafPort** is always the same as the window being zoomed.

Line 214 erases the window contents for reasons similar to the ones that apply when resizing windows. To see whether you agree this line is necessary, change it to a comment and add **Pause**; statements as you did before. Run the program and observe what happens to the scroll bar outlines when you zoom windows in and out.

Toolbox procedure **ZoomWindow** (215) takes three parameters: the window to zoom (**whichWindow**), a value (**partCode**) that tells whether to zoom in or out, and a parameter which, if **TRUE**, tells **ZoomWindow** to activate the window. **FALSE** allows programs to zoom inactive windows (those covered by others), but not to activate them. To do this in MacExtras, though, you'd have to rewrite **ZoomInOut** to include a similar Boolean parameter and pass it to **ZoomWindow** at line 215.

PROCEDURE CloseDAWindow;

When a desk accessory is the frontmost window, call **CloseDAWindow** (220–237) to close it. You'll do this usually to close accessories by choosing the File menu's Close command as well as by clicking the window close button. You can also call it in your program's shut-down routine, although the system closes desk accessories automatically when programs end and this step usually is not necessary. ApShell calls it in **DoClose** when it discovers that the front window doesn't belong to the program.

The procedure works by type casting the result of function **FrontWindow** to a **WindowPeek** data type (233). Because **FrontWindow** returns a **WindowPtr** result (pointing to a **GrafPort** record), to get to the window record's fields requires converting to a **WindowPeek** pointer. Line 234 extracts the **windowKind** field from that record, passing the number to toolbox procedure **CloseDeskAcc**.

**PROCEDURE GetPortSize(VAR width, height :
INTEGER);**

This procedure complements QuickDraw's **PortSize** routine, which changes the current **GrafPort**'s width and height. **GetPortSize** returns the current width and height—a job that programs often need to do.

It works by examining **thePort** (TYPE **GrafPtr**), a global variable (247) available to all programs that use QuickDraw. This pointer addresses the current **GrafPort**, telling QuickDraw routines in which port to draw. You change it when you call **SetPort**.

Notice that the port's width is simply its right coordinate value minus its left (249). Similarly, its height is its bottom minus its top (250). This always works because, as you recall from Chapter 3, coordinate values fall between pixels, never on the columns and rows. Therefore, subtracting two coordinate values always equals the number of pixels between.

**PROCEDURE CalcControlRects(whichWindow :
WindowPtr; VAR hBarRect, vBarRect, gbRect :
Rect);**

You saw this procedure (255–278) in action in ApShell's **DrawScrollBars** procedure. It calculates rectangles that encompass the horizontal (**hBarRect**), vertical (**vBarRect**), and grow box (**gbRect**) areas in a window. It assumes that the window actually has such items—you wouldn't call it if it doesn't.

Pass the window pointer in **whichWindow**. The procedure uses that window's **portRect** to calculate the three rectangles in the large **WITH** statement (261–277). The order of these statements is critical—don't change it.

```
FUNCTION TextHeight( wPtr : WindowPtr ) :
  INTEGER;
```

The shell doesn't use this function, but later examples do. It calculates the exact height of the current display font by passing a **FontInfo** record to **GetFontInfo** (293) and then adding the **ascent**, **descent**, and **leading** fields in that record to calculate the font's height in pixels. The function is most useful to avoid chopping text lines in half at window bottoms. Notice that **GetFontInfo** works on the current window and, therefore, **TextHeight** saves and restores the current **GrafPort** in the usual way.

```
PROCEDURE CenterString( h, v, w : INTEGER;
  s : Str255 );
```

Use this procedure (300–310) to center a string inside a window's borders or inside any other rectangular area. For example, you could center several strings in the upper quadrant of a window—you don't have to center them inside the full window width.

Set parameter **h** to the left side of the width in which you want to center text. Set parameter **w** to the width. To center a string between window horizontal coordinate values 20 and 100, set **h** to 20 and **w** to 80. Set parameter **v** to the vertical coordinate where you want text to appear. Remember that **QuickDraw** draws text with the base line—the line on which capital letters normally sit—and that lower-case letters like **p** and **q** extend below this line. The next procedure demonstrates how to use **CenterString**.

```
PROCEDURE DisplayAboutBox;
```

ApShell calls **DisplayAboutBox** (313–376) when you choose the Apple menu About ApShell command. It reads the string list with ID 1 from the program's resources into string array **messages** (340). A **FOR** loop loads each string resource (344–345), calling **GetIndString** (get indexed string).

To demonstrate how to create windows that don't have corresponding resource templates (as the ApShell window does), lines 347–351 first initialize a rectangle **wr** by copying the Macintosh **screenBits.bounds** values and then shrinking those values with **InsetRect** (348). This reduces the window size by 100 horizontal and 75 vertical pixels in from each border, centering the new window in the screen—no matter how large that screen is. (Remember that future Macintoshes might not have the same screen dimensions. Try never to hard-wire coordinates into your program. Always use coordinates relative to **screenBits.bounds**.)

Lines 349–350 create a new window by calling **NewWindow**, passing the address of a local **WindowRecord** (**wRec**), a **WindowPtr** (**wp**), a null string (' ') for the title, and the value **TRUE** indicating the window is to be visible immediately.

Following this is the constant **altDBoxProc**—representing a plain window with a shadow border along the left and bottom edges. The parameter **POINTER(-1)** tells the Window Manager to place the window in front of all others. The next parameter, **FALSE**, indicates that this window has no close box in the upper left corner. Finally, the last parameter, **0**, is a reference value, which in this case is meaningless.

As you can see from all of this complexity, explicitly creating windows in programs is more difficult than using resource templates as in ApShell. But this book wouldn't be complete if it didn't have at least one example of how to create windows the hard way. As a project, you might want to modify MacExtras to use a window template instead.

The statements at 355–373 change the current **GrafPort** to the new window and display the six resource strings. Lines 358–359 and 362–363 select fonts and text sizes. This displays the first string (the program title) in the system font, better known as Chicago, using a point size of 12, approximately 12/72 inches tall. (Point sizes are in 1/72-inch increments and it's customary not to reduce such fractions. The correct value is 12/72, not 1/6.) This makes the title come out in bold, Chicago's normal look. The other five strings display in the Geneva font, the default for most Macintosh text, but in a smaller 9-point size that looks good for this commercial message about the program.

The calls to **CenterString** (364–368) center the strings in the window. **Pause** (370) waits for you to click the mouse button. Notice that **CloseWindow** at line 371 erases the window and internal variables that the Window Manager creates for its own purposes. Line 372 finishes the procedure by restoring the current **GrafPort** to its original setting.

You might want to display this same message when your program begins rather than waiting for people to choose the About Program command. In fact, because all Mac programs look similar with menu bars and windows, it's sometimes hard to tell which program is running. To display a message when the program begins, insert a call to **DisplayAboutBox** in ApShell's **Initialize** procedure between lines 378 and 379. (You also have to add a semicolon at the end of 378.) This displays the program's commercial message at the beginning, which goes away as soon as you click the mouse button. If you purchased the disks, you'll discover that most examples operate this way. If you *don't* want this feature, remove the call to **DisplayAboutBox** from the program's **Initialize** procedure.

```
PROCEDURE DoAppleMenuCommands ( cmdNumber :  
    INTEGER ) ;
```

The final MacExtras procedure handles commands for the Apple menu, including displaying the About Program box and activating desk accessories. Because this menu rarely changes, although it might list different accessory names, it's best to keep it in a unit like MacExtras rather than recompiling it over and over for each new program.

Lines 389–391 display the About Program box if the **cmdNumber** equals constant **AboutCmd**. Otherwise, lines 393–401 open a desk accessory by first calling **GetItem**, which returns **daName** equal to the menu command name, and passing that name to **OpenDeskAcc** (399), which does the actual work. This function returns a result code, called the *driver reference number*, which the program ignores. (A desk accessory is known as a driver, a term that usually and probably more correctly refers to code that runs, or drives, printers and other devices.) In this case, the result code is meaningless and you always ignore it after calling **OpenDeskAcc**.

MacExtra's Main Body

A unit's main body runs before programs that use the unit. Some units don't have main bodies, but MacExtras does—executing a very important sequence of actions. Lines 409–416 initialize the Macintosh tools that ApShell and other programs use. Because nearly every Macintosh program starts with these same initializations, it's best to stuff them into a unit like MacExtras and forget about them.

The program comments tell you which toolset each line initializes. Three statements, however, might not be so clear. Line 409 passes the address of global **GrafPort** variable **thePort** to **InitGraf**. This must be the first step in any program that uses QuickDraw routines.

Line 414 initializes the Dialog Manager, which Chapter 6 explains in more detail. The **NIL** pointer value tells the Dialog Manager that this program has no *resume* procedure, one that the operating system calls upon receiving a fatal system error. Normally, you don't need to install such a procedure and can simply pass **NIL** as the value to **InitDialogs**.

Line 416 calls **FlushEvents**, passing constant **everyEvent** and a zero, called the *stop mask*. If instead of zero you passed a constant representing some other event kind, **updateMask** for instance, then **FlushEvents** would remove pending events only up to the first event of that kind. The zero tells it to remove all events in the queue. Together, the two parameters completely flush the event queue, removing any stray keypresses or mouse clicks that might be lurking there, waiting to surprise your program. (Seriously, this is important. Someone might click the mouse before a program gains control. The **FlushEvents** call at line 416 throws such extraneous events out.)

Windows, Text, and Scroll Bars

Nearly everything that happens in a fully charged Macintosh program takes place inside windows. You write text and display graphics in windows. You drag, resize, open, and close them. And, if you're like most people, you do all of those operations almost without thinking.

As the previous chapter demonstrates, adding windows to programs is easy because built-in ROM routines handle most of the details for you. Programs simply respond to certain events—mouse clicks, for example—and call the proper procedures to drag and resize windows and perform other standard operations.

But bare windows are useless—it's what's inside that counts. As you'll learn in this chapter, displaying text, graphics, and controls such as scroll bars in windows requires careful programming. You have to be concerned not only with the appearance of a window but with its structure in memory. For that reason, we'll begin with a look at the memory heap—a subject that many programmers warily approach as though it were a heap o' trouble. If you believe what many say, the heap is a mysterious never-never land—a place where variables fly around like Peter Pans, likely to be swallowed by the crocodile Memory Manager, never to be seen again.

In truth, the heap is nothing more than a large area of memory that follows specific organizational rules. The trouble is that many programmers misunderstand these rules, causing themselves plenty of unnecessary anguish—a bunch of Captain Hooks, nervous at every tick of the microprocessor clock. You won't be one of them if you understand how the heap works, as the next section explains.

HEAPS ARE FOR KEEPS

The heap is where programs keep most things in memory, including the program itself. Desk accessories take up heap space and font images reside there, as do many other variables and structures. In fact, it's almost ridiculous to describe

the many things you might find on the heap—practically everything in memory is there at one time or another.

To best understand this all-purpose heap, visualize memory as a block of empty space with boundaries that rise and fall according to how many objects you throw in. Figure 5.1 illustrates the idea with an over-simplification of Macintosh memory. At the bottom is space that the operating system reserves for its own purposes. Above that is the heap. At the top is the stack, expanding downwards toward the heap, which rises up to meet it. Both compete for the free space in between. In general, the heap contains data and code while the stack contains local variables—the ones you declare in Procedure and Function **VAR** sections—plus information about the running program. For example, when procedures call other procedures, the stack keeps information about who called whom and about the locations of variables.

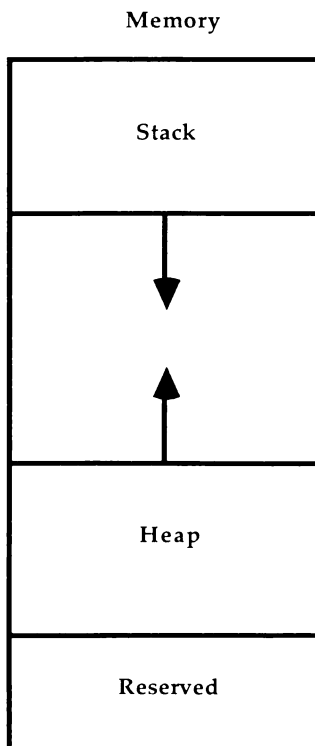


Figure 5.1 In memory, the stack grows down to meet the heap, which grows up. Both areas compete for free memory in between. The system reserves a portion of memory for its own use below the heap.

Because it's tied to the operation of a program, the stack grows and shrinks according to the order in which routines run, expanding up and down like a stack of pancakes. (In this case, it's an upside-down cake, but never mind.) Contrasting that action, the heap grows to accommodate the objects you tell it to keep. Unlike the stack, the heap may grow larger, but it never shrinks. Once the heap expands to a certain size, it stays there like a thermometer that records the high temperature, pushing the marker up and up but never letting it fall.

Inside the heap are blocks of memory, some with program variables and others empty, available for programs to use. When the heap contains no empty blocks, it expands to hold new objects. Otherwise, it uses as many empty blocks as it can, keeping its top as low as possible and, therefore, keeping total memory use to a minimum.

As you can see in Figure 5.1, if the top of the heap grows higher than the bottom of the stack, both areas overlap, a dangerous condition to avoid at all costs. To detect this problem, the operating system “sniffs” 60 times a second for the smoke released by a stack/heap collision. If this stack sniffer detects any smoke—in other words, if the stack bottom is ever lower than the heap top—it calls the system error handler, displaying a bomb box with error 28, stack overflow.

You must prevent this error from occurring. There is no recovery. In fact, you must go further, preventing even its possibility. The stack sniffer finds out about collisions on the average only 1/120 of a second *after* they occur. That's eons of computer time, enough for the stack to expand into the heap, destroy variables, and then pull itself back up before the sniffer checks for the error! To avoid collisions, follow these rules.

- Keep procedure nesting to a minimum and use only small local variables, particularly in recursive routines
- Always place big variables—arrays and large record structures, for example, on the heap
- Dispose the objects your program no longer needs. This returns their memory to the heap, making it available for other uses
- Avoid declaring simple pointers to objects. Use handles whenever you can

The last of these rules is the most important. A handle is a special kind of pointer that lets the Memory Manager organize the heap to your program's benefit. By using handles, you take advantage of the manager's ability to move memory blocks and to use efficiently all available memory before expanding the heap. This action tends to keep the heap top as low as possible, reducing the likelihood that it ever will bump into the stack.

All About Handles

As you may know, a Pascal pointer points to an object in memory. To create such objects, you declare pointers to their data types and use a **New** statement to

allocate memory for variables of those types. The pointers hold the addresses of variables in memory. If you're a little rusty on pointers, the following example helps make them clear.

```

TYPE
  OneRec =
    RECORD
      a, b : INTEGER
    END;

VAR
  p : ^OneRec;

BEGIN
  New( p );
  p^.a := 10;
  p^.b := 20;
  Dispose( p )
END.

```

The statement **New(p)** allocates memory on the heap for a variable of type **OneRec**, a record with two integer fields, **a** and **b**. Then, two statements assign 10 to the first field (**a**) and 20 to the other (**b**). The caret *dereferences* the pointer, telling Pascal to refer to the object to which the pointer points—not to the pointer itself as an object. (Remember that pointers are variables. They're special because they point to other variables, in this example, a record with two integer fields.) The last statement disposes the memory that **p**[^] occupies, making that memory available for other uses.

The trick to good Macintosh memory management is to understand how the standard Pascal method works and then forget it. *Never* use **New** to allocate memory for pointers as this example demonstrates. Although it works, it's a poor memory

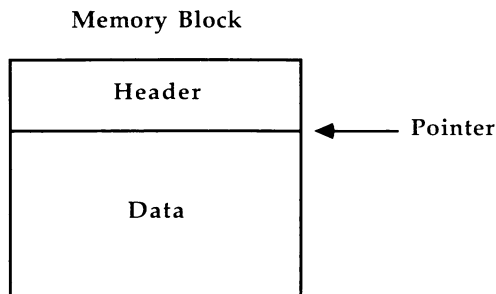


Figure 5.2 All memory blocks begin with a header, which the Memory Manager uses to organize the heap. Program pointers address data in blocks, not their headers.

management technique. To understand why requires knowing how the Memory Manager keeps track of memory areas, or blocks, on the heap.

Every time you request heap space for a program variable, the Memory Manager creates a block containing the information it needs to organize and locate all blocks in memory plus enough room to store your program's variables. Every such block has the organization in Figure 5.2.

The memory block header contains information that records the block's size along with other facts such as whether the Memory Manager can move it to make room for other blocks. The header is always eight bytes long. You don't need to know the header's exact contents and you'll probably never directly use it. Don't even assume that it will remain at eight bytes in all future Macintosh incarnations. Just be aware that every block—whether used or not—has a header.

Following the header are the bytes that belong to your dynamic object; in other words, the variable in memory that a pointer addresses. (The object is *dynamic* because you create it explicitly when your program runs and dispose the space later.)

When you create a dynamic variable with **New**, the Memory Manager reserves a *non-relocatable* memory block with a header and space to hold your data. Eventually, after many such statements, your heap looks like Figure 5.3 with many

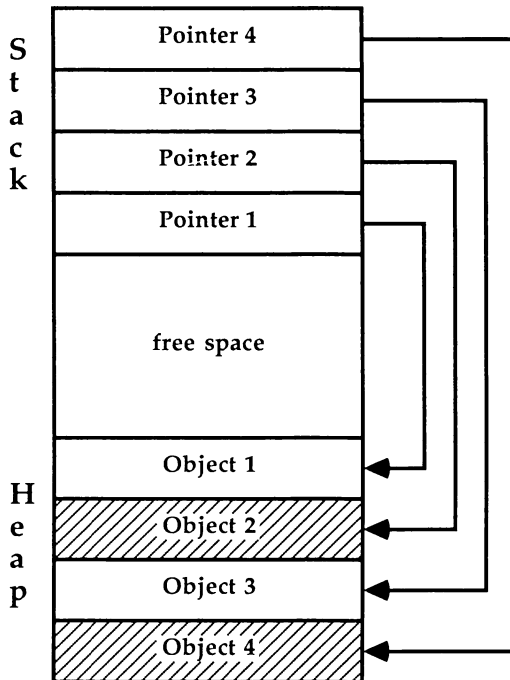


Figure 5.3 Simple pointers on the stack address objects on the heap, a technically correct though poor memory management practice.

pointers and blocks. Notice that the pointers themselves are local program variables on the stack. (They also could be global variables.)

Problems with this arrangement begin appearing when you dispose objects in the middle of other non-relocatable blocks. The disposals leave holes in the heap—the shaded areas in the figure. These holes are not lost to the Memory Manager and it can use them for other objects. But new objects can be no bigger than the largest hole available. To create an object larger than that requires expanding the heap, taking up more of the computer's free space, and increasing the danger of a stack/heap collision.

Having many disposed non-relocatable objects in memory can cause a condition called *fragmentation*. The objects fragment the heap into a gulf stream of immovable islands. Consequently, you can build variables no larger than the available land space.

Relocatable memory blocks help prevent this condition because the Memory Manager can move them around to compact unused islands into a larger land mass. Only if all such areas combined are too small to hold your new object will the heap expand further into the ocean of available free space. But looking again at Figure 5.3, if the manager shifted objects in order to combine numbers 2 and 4, the program might lose the locations of 1 and 2, resulting in a problem that Figure 5.4 demonstrates.

Pointers 3 and 4 now are wrong! The heap is nicely arranged with its free space and disposed objects together, making the most of available memory. All used objects are as far down in memory as possible. But the pointers on the stack no longer point to the correct objects and the program would fail to operate.

To fix this problem and to let programs find relocatable objects no matter where the Memory Manager moves them, requires the help of a *master pointer*. Instead of pointing directly to objects in memory, special variables called *handles* point to these master pointers, which in turn locate the real variables on the heap. Because the Memory Manager never moves master pointers and because it automatically adjusts them when it moves memory blocks to which they point, handles always locate objects while making the most of heap space.

Figure 5.5 shows how handles, master pointers, and relocatable objects cooperate. Instead of simple pointers, the program creates handles as local variables on the stack or as global variables permanently in memory. These handles point to master pointers far down on the heap. The master pointers in turn point to the actual objects. When the Memory Manager moves an object, it automatically adjusts the master pointer to the correct value, locating objects in their new positions.

Notice that master pointers to disposed objects (numbers 2 and 4 in the figure) are **NIL**, represented by electrical grounding symbols. But the handles still address the same master pointers. This situation happens when memory blocks are not only locatable but *purgeable*, meaning you give the Memory Manager the right to remove the objects to which they point. For example, if you mark Object 3 purgeable, the Memory Manager can reuse the space it occupies. If it does, it then sets Master Pointer 3 to **NIL**. Before programs attempt to use purgeable memory blocks,

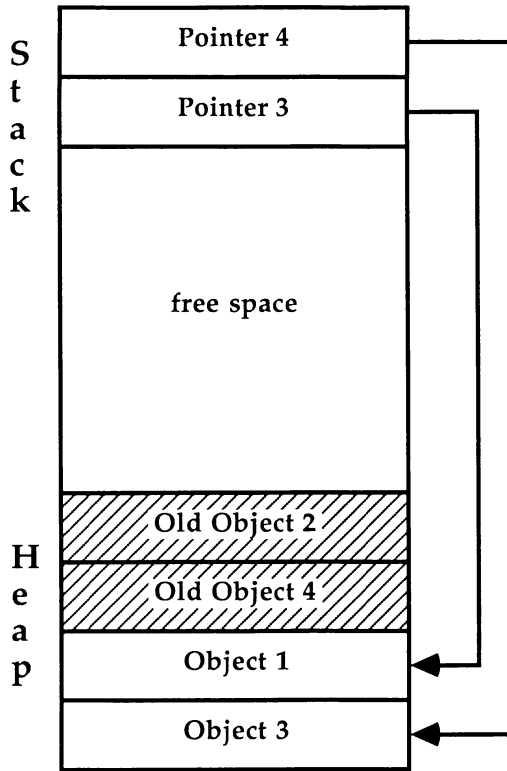


Figure 5.4 If objects were rearranged to combine their memory with the free space between the stack and the heap, simple pointers would lose track of the objects to which they refer.

therefore, they must check whether the Memory Manager had previously removed them from memory. If the master pointer is **NIL**, then the program must recreate that object—perhaps reading it from disk. This is the way the operating system loads various program resources such as the definitions for clickable buttons, scroll bars, and even portions of the program’s code. When you mark such items purgeable, as you did in Chapter 4, you are telling the Memory Manager it is free to remove them from memory when it needs more room.

You need to understand one more fact about master pointers. In Figure 5.5, the four pointers at the bottom of the heap must never move. If they did, the program’s handles couldn’t find them—just as it couldn’t find the movable objects in Figure 5.4. For this reason, master pointers always reside in non-relocatable memory blocks. You have a lifetime warranty from the operating system that your master pointers will stay where they are.

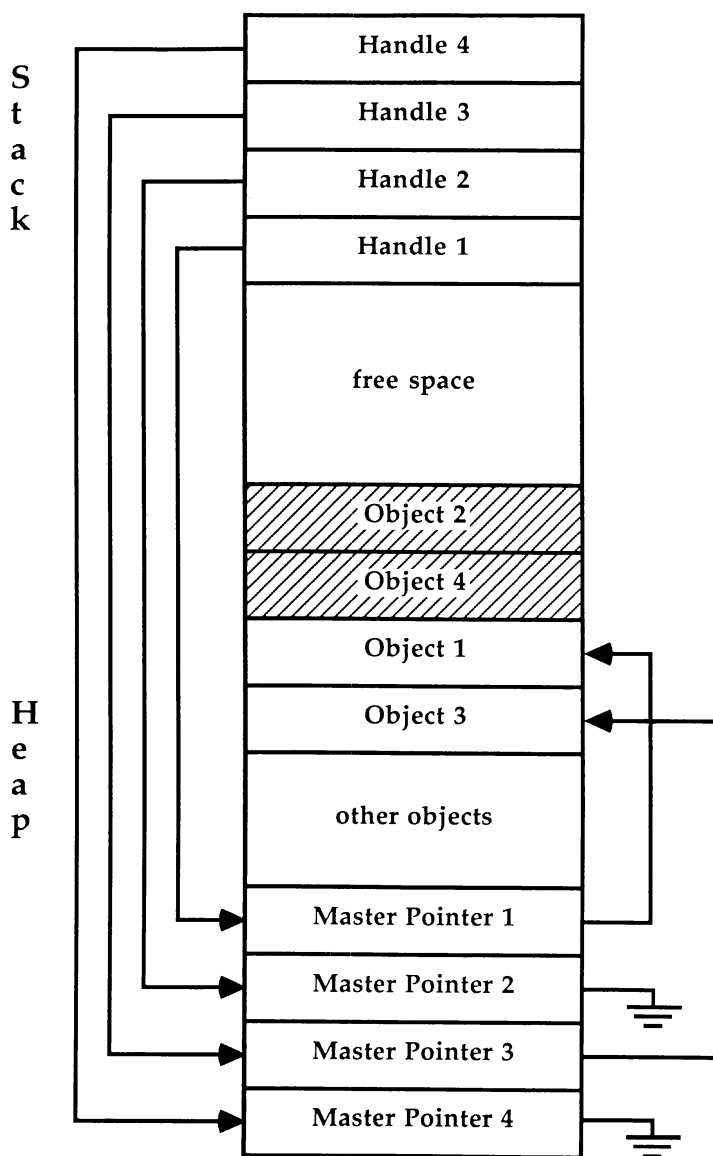


Figure 5.5 Good Macintosh memory management uses handles instead of pointers. With this method, handles on the stack address master pointers on the heap, which in turn address objects. This lets the Memory Manager move objects to better use available memory but still allow programs to find their data.

Unfortunately, because master pointer blocks are immovable, they too can fragment the heap—reducing the advantage of using handles in the first place. To make certain that never happens, the Memory Manager attempts to place master pointer blocks as low on the heap as it can. At the start, all programs have a group of master pointers available for new handles. (Currently, the operating system creates 64 master pointers at a time, placing them all in one 256-byte non-relocatable block. But don't rely on these numbers. They might change in the future.) If a program creates more than that many handles, the Memory Manager automatically allocates an additional block of master pointers.

Meanwhile, if you created immovable objects on the heap, above the original block of master pointers, the new masters go above your objects. Later, if you dispose of those objects, the master pointer blocks themselves fragment the heap, as Figure 5.6 shows. To prevent this situation from occurring, programmers typically add several **MoreMasters** statements to their initialization procedure. Each call to this routine creates an additional block of master pointers. By calling it early in the program—before it creates any objects on the heap—all master pointer blocks are as low as possible, most likely compacted together near the bottom of the heap, the ideal arrangement.

How many **MoreMasters** statements do you need? Some people use six or eight such calls even in programs that use only one or two handles. Such overkill might actually waste more memory than the worst possible fragmentation the program could produce! A better solution, then, is to follow these suggestions:

- Calculate the maximum number of handles your program will use at one time. Round this number up to the nearest multiple of 64, divide the result

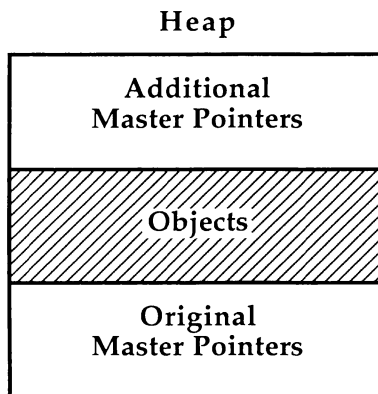


Figure 5.6 The Memory Manager creates blocks of master pointers as needed, an action that can fragment the heap as shown here. To prevent this condition, call toolbox procedure **MoreMasters** to create all the master pointers your program is likely to need.

by 64, and call **MoreMasters** that many times. This creates a few extra handles for programs such as desk accessories that share heap space with your routines. Remember that toolbox routines create their own handles for various reasons.

- Use a debugger like TMON or MacsBug to examine the heap after stepping your program through its paces. Add **MoreMasters** calls until all master pointer blocks are together near the bottom of the heap. If you cannot calculate beforehand how many handles your program will create, you have little choice but to use this trial-and-error technique.
- Use handles to relocatable objects for all dynamic variables. This way, the Memory Manager will move your objects upward when it creates new master pointer blocks, placing those blocks as low as it can on the heap. If you do this, you don't have to call **MoreMasters** to reserve blocks ahead of time.

Although the last suggestion is probably best, and it lets the Memory Manager organize memory rather than forcing the job onto your program, it doesn't always work. One reason the idea fails is that window records must be in non-relocatable memory blocks, an unbreakable if unfortunate rule. It also fails when programs temporarily lock relocatable blocks to prevent the Memory Manager from moving them for a limited amount of time. If the Memory Manager creates new master pointer blocks while intervening relocatable blocks are locked, the heap can become badly fragmented. Both of these cases defeat the manager's attempts to place non-relocatable master pointer blocks low on the heap. Therefore, it's probably a good idea to call **MoreMasters** at least a few times in any program that uses many handles.

Molding Your Own Handles

Creating your own handles is easy although many people find the subject confusing at first. The best plan is to have a good mental image of the heap, as in Figures 5.1 through 5.6. In particular, refer to Figure 5.5 on page 188. The following examples show how to create a program that follows this illustration.

First, define an object. I'll use a simple record structure, although you can create any other kind of relocatable object, arrays, sets, strings, and so on. Here is the data type.

```
TYPE
  OneRec =
    RECORD
      a, b : INTEGER;
      r : REAL;
      s : STRING[40]
    END;
```

Each **OneRec** record has four fields: two integers, a real number, and a 40-character string. You could create local variables of this type, but then they would take up valuable stack space. Instead, it's better to create variables on the heap as relocatable objects. This requires two pointers:

```
TYPE
  OneRecHand = ^OneRecPtr;
  OneRecPtr = ^OneRec;
```

OneRecPtr is a simple pointer to a **OneRec** object. This is the master pointer—the one that actually addresses objects in memory. **OneRecHand** is a pointer to a **OneRecPtr**. This is the handle, the variable that the program creates to address the master pointer. Be sure you understand that the handle points to the master pointer, which points to the actual object. With this design, we're ready to create the program's variables, the four handles in Figure 5.5. To do that, you start with this global **VAR** declaration:

```
VAR
  Handle1, Handle2,
  Handle3, Handle4 : OneRecHand;
```

This completes the program's declarations. You now have a data type (**OneRec**), a master pointer to that type (**OneRecPtr**), a handle to a master pointer, (**OneRecHand**), and four variables, **Handle1** through **Handle4**. These variables correspond with those on the stack in Figure 5.5. If the variables are global, they actually reside above the stack in the application's global space. In either case, the organization is similar. Notice that everything but the handle variables are **TYPE** declarations, which take no room in memory. They merely define structures that don't yet exist.

The next step is to create space on the heap for objects defined by the earlier **TYPE** declarations. You do that by calling the Memory Manager function **NewHandle**, which reserves space for an object (plus the header bytes that go with all memory blocks), assigns the address of that space to a master pointer, and returns the address that becomes the program's handle. As shown here, one step accomplishes all of these actions.

```
Handle1 := OneRecHand ( NewHandle ( Sizeof ( OneRec ) ) );
IF Handle1 = NIL
  THEN DoErrorRoutine;
```

Placing **NewHandle** inside **One RecHand**'s parentheses converts the plain handle data type to that of **Handle1**—an example of *type casting*. A variable of one type is recast to another. Repeat that same step for each of your handles. Notice that **Sizeof** passes **OneRec**'s size in bytes to **NewHandle**. The Memory Manager reserves that much space plus room for the block header. You never have to leave

space for this header—as far as your program is concerned, objects are no bigger than **Sizeof** reports. If **NewHandle** cannot create space for an object, it returns **NIL**. As the example shows, you should test for this problem immediately after calling **NewHandle**. There's no need to check beforehand whether enough memory exists for new objects.

You now have a heap arranged similarly to the illustration in Figure 5.5. (Objects 2 and 4 are not yet disposed, of course.) To assign values to variables, follow the handles to the master pointers, which in turn locate the objects in memory. For example, to add values to object 1, you could write:

```
WITH Handle1^^ DO
BEGIN
  a := 10; b := 20;
  r := 3.14159;
  s := 'This is a string'
END; { with }
```

The two carets dereference the handle twice. As the standard Pascal example that began this section illustrates, a single dereference locates the object to which the pointer points. Therefore, the first caret finds the object that the handle addresses—in other words, the master pointer. The second caret locates the object to which that pointer points—the actual object in memory.

Some people would have you believe that you must never double dereference handles in **WITH** statements as this example shows. Nonsense. You must dereference handles in order to get to your objects and, by using a **WITH** statement, you generate code that runs faster than the alternative many programs use. For example, the following statements work too hard to achieve the same result:

```
Handle1^^.a := 10;
Handle1^^.b := 20;
Handle1^^.r := 3.14159;
Handle1^^.s := 'This is a string';
```

Every **Handle1^^** expression forces the compiler to generate code to perform two address calculations. By using **WITH**, you let the compiler generate code to calculate the address a single time and then use that same address for each of the following assignments. Why, then, do so many Macintosh programmers do it the hard way?

The answer is they know the Memory Manager might move relocatable objects in order to make room for others. If that should happen in between the time of your **WITH** statement and an assignment, the address calculation would become invalid. This happens because **WITH** makes a *copy* of the master pointer and then uses that copy to locate your objects. If, in the meantime, the Memory Manager should change the master pointer's value—which it certainly will do if it moves the associated memory block—the copy of that address might become wrong and the program could reference an object after it moves to some other location.

Out of fear of this happenstance, people avoid using **WITH** statements as though they were viral bugs that plague only Macintosh programmers. But such fears are groundless. The Memory Manager guarantees to you that it will move objects only at specific times, and only when you call certain routines in the toolbox. Objects will never move at any other time. Period. Volumes 3 and 4 of *Inside Macintosh* list all of these routines. As long as you don't call any of them, relocatable objects will not move.

As with all perfect warranties, though, there's a catch. If you call a procedure that calls other routines that in turn end up calling one of the routines in the list, the Memory Manager might shuffle objects around even though it appears as though such an event could never happen. For example, suppose you call a procedure and pass it one of the fields from your object. If you write:

```
ProcessValue( Handle2^^.r );
```

and if **ProcessValue** declares its parameter as a variable, a fact that is not obvious from the statement that calls it, Pascal passes a *copy* of the address of field **r**. If **ProcessValue** then calls one of the memory-shuffling routines, that address might become wrong if the object containing **r** moves. This can never happen when passing parameters by value to routines, in which case Pascal makes a copy of the value itself—not the value's address in memory. In other words, if the procedure you call declares **VAR** parameters, be wary of double dereferencing handles in calls to that procedure. If the procedure declares only value parameters, even if it calls one of the memory-shuffling routines, you have nothing to worry about.

What do you do, then, when you have to use the address of objects in relocatable blocks? There are two answers. One, make a copy of the value you want, process that value, and then reassign it to the object on the heap. If you have a **REAL** number variable **tempR**, you could write:

```
tempR := Handle2^^.r;
ProcessValue( tempR );
Handle2^^.r := tempR;
```

Even if the object at **Handle2^^** should move during the call to **ProcessValue**, the third statement locates it properly by again double dereferencing the handle. This always works. Objects never move during simple assignments.

The second solution is to lock the object on the heap, temporarily preventing the Memory Manager from moving it. Even if you call one of the memory-shuffling routines, your locked objects will not move. The program now becomes:

```
Hlock( Handle( Handle2 ) );
    ProcessValue( Handle2^^.r );
Hunlock( Handle( Handle2 ) );
```

Hlock locks the relocatable block associated with a certain handle. **Hunlock** unlocks that block, allowing the Memory Manager to move it once again. Because

both procedures take a plain vanilla **Handle** data type, you have to typecast your handles to that generic type as shown.

Although this solution is attractive and avoids double dereferencing handles more than once, you pay a heavy price for that advantage. Because you have locked the relocatable memory block, you've defeated the purpose of using handles. Even worse, you risk fragmenting the heap. If the Memory Manager needs more room, it will not be able to move your locked block and might therefore have to expand the heap needlessly. For these reasons, it's best to avoid locking relocatable blocks, even temporarily.

Disposing Handles

When you're finished using dynamic objects, always dispose their handles. This releases space the objects occupy on the heap, making it available for other uses. You have to do nothing special to ensure that the Memory Manager reuses disposed memory. It will as long as you dispose your handles this way:

```
DisposeHandle( Handle( Handle1 ) );
DisposeHandle( Handle( Handle2 ) );
```

Calling **DisposeHandle** gives the Memory Manager permission to reuse the memory block associated with a handle. As when you create handles, you have to typecast your variables to generic **Handle** types when passing them to this routine. After disposing a handle, the master pointer it addresses is again available for future calls to **NewHandle**. For this reason, it may be a good idea also to set your handles to **NIL** as follows:

```
Handle1 := NIL;
Handle2 := NIL;
```

This is optional, but what's important is that you never use a handle after disposing it. You can assign to it the result of **NewHandle** to create a fresh relocatable object on the heap, but that's it. This is a doubly important rule because the Memory Manager does not change your handle's value after you dispose its memory block. It's up to you to guard against accidentally reusing disposed handles.

MULTIPLE WINDOWS

ApShell in Chapter 4 opens a single test window. Although some programs need only one, most allow many windows at one time. The next example demonstrates a good method for keeping track of multiple windows and shows how to deal with the special memory management problems this presents.

Type in Listing 5.1, save as MULTIWIND.R, and compile with RMaker. Also

Listing 5.1. MULTIWIND.R

```

1: *-----*
2: * MultiWind.PAS resources -- Compile with RMaker *
3: *-----*
4:
5: Programs:Windows.F:MultiWind.RSRC      ;; Send output to here
6:
7:
8: *-----*
9: * About box string list *
10: *-----*
11:
12: TYPE STR#                                ;; String list resource
13:     ,1 (32)                             ;; ID and attribute (purgeable)
14: 6                                         ;; Number of strings that follow
15: Multi Window Demo                       ;; Program name
16: by Tom Swan                             ;; Author
17: Version 1.00                            ;; Version number
18: (C) 1987 by Swan Software               ;; Copyright notice
19: P. O. Box 206, Lititz, PA 17543        ;; Address
20: (717)-627-1911                         ;; Telephone
21:
22:
23: *-----*
24: * The Apple Info menu *
25: *-----*
26:
27: TYPE MENU
28:     ,1                                   ;; Menu ID number to use in program
29:     \14                                 ;; Bitten-apple graphics symbol
30:     About Multi...                     ;; The command as shown in menu
31:     (-                                 ;; Divider line between command and DAS
32:
33:
34: *-----*
35: * The File menu *
36: *-----*
37:
38: TYPE MENU
39:     ,2
40: File
41:     New /N
42:     (Close
43:     (-
44:     Quit /Q
45:
46:
47: *-----*
48: * The Edit menu *
49: *-----*
50:
51: TYPE MENU
52:     ,3
53: Edit
54:     (Undo /Z
55:     (-
56:     (Cut /X
57:     (Copy /C
58:     (Paste /V
59:     (Clear
60:
61:
62: * END

```

type in Listing 5.2 and save as MULTIWIND.PAS. To save space here, lines 118 and 194 tell you to insert programming from ApShell (Listing 4.1). Compile MULTIWIND.PAS. When you run the program, choose New from the File menu to open windows. Each one overlaps the previous, similar to the way the Turbo Pascal editor operates.

Listing 5.2. MULTIWIND.PAS

```

1: {$O Programs:Windows.F: }           { Send compiled code to here }
2: {$R Programs:Windows.F:MultiWind.Rsrc} { Use this compiled resource file }
3: {$U-}                               { Turn off standard library units }
4:
5:
6: PROGRAM MultiWind;
7:
8: (*
9:
10:  * PURPOSE : Multiple-windows demo
11:  * SYSTEM  : Macintosh / Turbo Pascal
12:  * AUTHOR   : Tom Swan
13:
14:  *)
15:
16:
17: {$U Programs:Units.F:MacExtras }      { Open this library unit file }
18:
19:
20:     USES
21:
22:         Memtypes, QuickDraw, OSIntf, ToolIntf, PackIntf, MacExtras;
23:
24:
25:
26:     CONST
27:
28:         FileID           = 2;      { File menu Resource ID and commands }
29:         NewCmd            = 1;
30:         CloseCmd          = 2;
31:         {-----}
32:         QuitCmd           = 4;
33:
34:         MaxWindow        = 8;      { Maximum windows open at once }
35:
36:
37:
38:     VAR
39:
40:         wPtrs : ARRAY[ 1 .. MaxWindow ] OF WindowPtr;
41:
42:         numWindows : 0 .. MaxWindow;      { Number of windows open }
43:
44:         quitRequested : BOOLEAN;          { TRUE if quitting }
45:
46:
47:
48: FUNCTION WindowIsOurs( wPtr : WindowPtr;
49:     VAR wIndex : INTEGER ) : BOOLEAN;
50:
51: { TRUE if the window at wPtr belongs to this program. If it is
52:  ours, then wIndex = its wPtrs array index. Otherwise, wIndex
53:  is meaningless. }
54:

```

```

55:  VAR
56:
57:      i : INTEGER;
58:
59:  BEGIN
60:      FOR i := 1 TO MaxWindow DO
61:          IF wPtrs[ i ] = wPtr THEN
62:              BEGIN
63:                  WindowIsOurs := TRUE;
64:                  wIndex := i;
65:                  Exit
66:              END; { if }
67:          WindowIsOurs := FALSE
68:      END; { WindowIsOurs }
69:
70:
71:  FUNCTION NextIndex : INTEGER;
72:
73:  { Return next available index in wPtrs array for creating new
74:  windows. Assumes numWindows < MaxWindow. Returns -1 if
75:  no array index available. }
76:
77:  VAR
78:
79:      i : INTEGER;
80:
81:  BEGIN
82:      FOR i := 1 TO MaxWindow DO
83:          IF wPtrs[ i ] = NIL THEN
84:              BEGIN
85:                  NextIndex := i;
86:                  Exit
87:              END; { for / if }
88:          NextIndex := -1          { Fail safe value }
89:      END; { NextIndex }
90:
91:
92:  PROCEDURE CalcWindRect( wIndex : INTEGER; VAR wRect : Rect );
93:
94:  { Calculate coordinate values for this window's bounds rectangle }
95:
96:  CONST
97:
98:      StartTop    = 43;      { Fixed upper right corner coordinate }
99:      StartLeft   = 16;
100:
101:  VAR
102:
103:      p, q : INTEGER;      { Temporary work variables }
104:
105:  BEGIN
106:      p := ( wIndex - 1 ) MOD 4;
107:      q := ( ( wIndex - 1 ) DIV 4 ) * 4;
108:      WITH wRect DO
109:          BEGIN
110:              top      := StartTop + q + ( p * 21 );
111:              left     := StartLeft + q - ( p * 4 );
112:              bottom   := screenBits.bounds.bottom - 5;
113:              right    := screenBits.bounds.right - 5
114:          END { with }
115:      END; { CalcWindRect }
116:
117:
118: << INSERT LINES 61-74 FROM APSHELL.PAS >>
119:
120:

```

(continued)

```

121: PROCEDURE DoNew;
122:
123: { Respond to File menu New command }
124:
125:   CONST
126:
127:       Visibility  = FALSE;    { Window not immediately visible }
128:       WindowID    = 8;        { Standard window with zoom box }
129:       HasGoAway   = TRUE;     { Window has a close box }
130:       RefCon      = 0;        { Window reference value (none) }
131:
132:   VAR
133:
134:       wPtr : WindowPtr;
135:       wTitle : Str255;
136:       wIndex : INTEGER;
137:       wRect : Rect;
138:
139:   BEGIN
140:       IF numWindows < MaxWindow THEN
141:           BEGIN
142:               wIndex := NextIndex;
143:               IF wIndex > 0 THEN
144:                   BEGIN
145:                       NumToString( wIndex, wTitle );
146:                       wTitle := Concat( 'Test Window # ', wTitle );
147:                       CalcWindRect( wIndex, wRect );
148:                       wPtr := NewWindow( NIL, wRect, wTitle, Visibility,
149:                                           windowID, POINTER(-1), HasGoAway, RefCon );
150:                       IF wPtr <> NIL THEN
151:                           BEGIN
152:                               (* SetWTitle( wPtr, wTitle ); *)
153:                               wPtrs[ wIndex ] := wPtr;
154:                               numWindows := numWindows + 1;
155:                               ShowWindow( wPtr );
156:                               EnableItem( fileMenu, CloseCmd );
157:                               IF numWindows = MaxWindow
158:                                   THEN DisableItem( fileMenu, NewCmd )
159:                               END { if }
160:                           END { if }
161:                       END { if }
162:                   END; { DoNew }
163:
164:
165: PROCEDURE CloseProgramWindow( wIndex : INTEGER );
166:
167: { Close this window in wPtrs array }
168:
169:   BEGIN
170:       DisposeWindow( wPtrs[ wIndex ] );
171:       wPtrs[ wIndex ] := NIL;
172:       numWindows := numWindows - 1;
173:       EnableItem( fileMenu, NewCmd )
174:   END; { CloseProgramWindow }
175:
176:
177: PROCEDURE DoClose;
178:
179: { Respond to File menu Close command }
180:
181:   VAR
182:
183:       wIndex : INTEGER;
184:

```

```

185: BEGIN
186:   IF WindowIsOurs( FrontWindow, wIndex )
187:     THEN
188:       CloseProgramWindow( wIndex ) { Close program's window }
189:     ELSE
190:       CloseDAWindow { Close desk accessory window }
191:     END; { DoClose }
192:
193:
194: << INSERT LINES 124-368 FROM APSHELL.PAS >>
195:
196:
197: PROCEDURE Initialize;
198:
199: { Program calls this routine one time at start }
200:
201:   VAR
202:
203:     i : INTEGER;
204:
205:   BEGIN
206:     SetUpMenuBar; { Initialize and display menus }
207:     quitRequested := FALSE; { TRUE on selecting Quit command }
208:     numWindows := 0; { No windows open }
209:     FOR i := 1 TO MaxWindow DO { Initialize window pointers array }
210:       wPtrs[ i ] := NIL; { to all NIL values. }
211:       DisplayAboutBox { Identify program }
212:     END; { Initialize }
213:
214:
215: FUNCTION QuitConfirmed : BOOLEAN;
216:
217: { The program's "deinitialization" routine. If someone chooses quit
218: command, this routine closes any open windows and tells the main
219: program loop whether it is okay to end the program now. }
220:
221: BEGIN
222:   IF quitRequested THEN
223:     WHILE FrontWindow <> NIL DO
224:       DoClose;
225:       QuitConfirmed := quitRequested
226:     END; { QuitConfirmed }
227:
228:
229: PROCEDURE DoSystemTasks;
230:
231: { Do operations at each pass through main program loop }
232:
233:   VAR
234:
235:     wIndex : INTEGER;
236:
237:   BEGIN
238:
239:     SystemTask; { Give DAs their fair share of time }
240:
241:     IF FrontWindow = NIL THEN
242:
243:       BEGIN { Set up menu commands for empty desktop }
244:
245:         FixEditMenu( FALSE );
246:         EnableItem( fileMenu, NewCmd );
247:         DisableItem( fileMenu, CloseCmd );
248:
249:       END ELSE
250:

```

(continued)


```

251:      IF NOT WindowIsOurs( FrontWindow, wIndex ) THEN
252:
253:          BEGIN { Set up menu commands for active desk accessory }
254:
255:              FixEditMenu( TRUE );
256:              EnableItem( fileMenu, CloseCmd )
257:
258:          END { else / if }
259:
260:      END; { DoSystemTasks }
261:
262:
263: BEGIN
264:
265:     Initialize;
266:
267:     REPEAT
268:
269:         DoSystemTasks;
270:
271:         IF GetNextEvent( everyEvent, theEvent ) THEN
272:
273:             CASE theEvent.what OF
274:
275:                 MouseDown    : MouseDownEvents;
276:                 KeyDown       : KeyDownEvents;
277:                 AutoKey        : { ignored };
278:                 UpdateEvt      : UpdateEvents;
279:                 ActivateEvt    : ActivateEvents
280:
281:             END { case }
282:
283:         UNTIL QuitConfirmed
284:
285:     END.

```

MultiWind Play-by-Play

MULTIWIND.R (1-62)

Notice that the resource text file for the program does not contain a template for the window design as in ApShell. You could certainly design multiple window programs that way, but in this case the program will create its windows on its own. Other than this omission, MULTIWIND.R is the same as ApShell's resource text.

MULTIWIND.PAS (1-44)

Line 34 declares the maximum number of windows MultiWind can open at one time. Change this value to allow as many windows as your program needs. The program creates window records on the heap rather than as local or global variables.

Array **wPtrs** (40) holds one **WindowPtr** for each possible window the program opens. Some programmers do this differently, declaring an array of **WindowRecord** variables instead of pointers to those records. Both methods are acceptable, but

remember that **WindowRecords** are large, each taking more than 150 bytes. For that reason, it's best to create them on the heap rather than permanently occupy memory as global variables or take away scarce stack space as local variables in procedures and functions.

Line 42 creates an associated variable, **numWindows**, which counts the number of windows now open. This makes it easy for the program to know when it reaches its maximum limit. When **numWindows** is zero, no windows are open.

WindowIsOurs to CalcWindRect (48–115)

WindowIsOurs (48–68) returns **TRUE** if parameter **wPtr** is the same as one of the pointers in array **wPtrs**. You call the function to determine if a window belongs to the program or to another process, most likely a desk accessory. (See **DoClose** 177–191 for an example.) If it returns **TRUE**, then parameter **wIndex** equals the index into array **wPtrs** for this window pointer.

Function **NextIndex** (71–89) returns the value of an available window pointer from array **wPtrs**. If no pointers are free, it returns **-1**, a situation that cannot occur if you check beforehand that **numWindows** is less than **MaxWindow**. Multi-Wind calls **NextIndex** when creating new windows in order to know where to store their pointers in the **wPtrs** array.

For each new window that it creates, the program calls **CalcWindRect** (92–115), which returns parameter **wRect** set to the window's initial boundary rectangle. The formula calculates overlapping rectangles so that at least some of each window's top and left borders are visible no matter how many windows you open at once.

DoNew to END (121–285)

Procedure **DoNew** (121–162) creates windows without using a resource template. Its four constants (127–130) make the program more readable (see 148–149). **Visibility** is **FALSE** to let the program create windows, adjust certain parameters, and then display them with a single call to **ShowWindow** (155). In this example, you can set **Visibility** to **TRUE** with no bad effect. But in many programs, you might want the opportunity to perform some action on windows after creating them but before displaying them the first time. Set **WindowID** (128) to 0 if you don't want a zoom box. To use other window styles, change this constant to one of the values in parentheses or to one of the underlined identifiers in Figure 5.7. If you change **HasGoAway** to **FALSE**, then windows won't have close boxes. (You can still close them with the File menu's Close command, though.)

To create a new window, **DoNew** first checks whether **numWindows** exceeds **MaxWindow** (140). As long as it doesn't, line 142 calls **NextIndex** to set local variable **wIndex** to the next available **wPtrs** array index. Lines 145–147 create a window title string and calculate the window's boundary rectangle by calling **CalcWindRect**. Then, lines 148–149 use window manager routine **NewWindow** to create the window record in memory and assign its address to **wPtr**. Unlike the method that

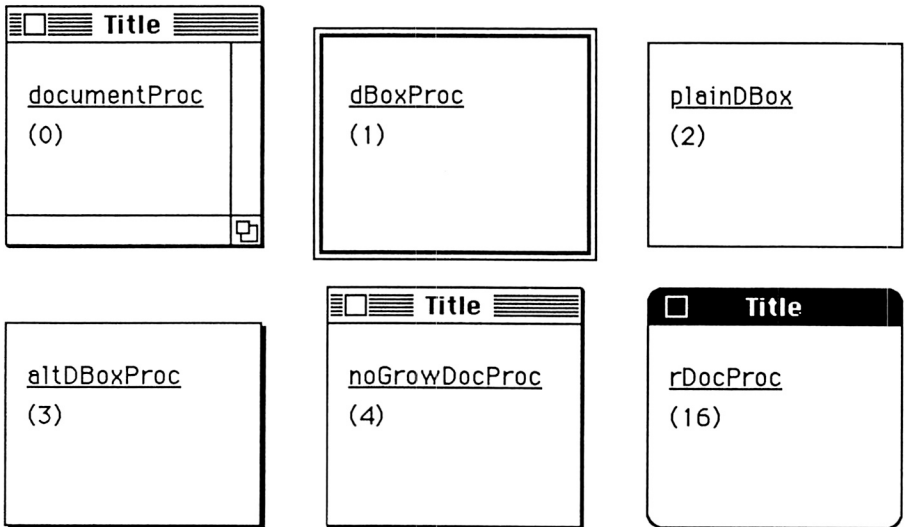


Figure 5.7 When designing windows, you have six styles to choose from. The underlined identifiers are constants with the values in parentheses. Missing are windows with zoom boxes (8), not a separate style but actually a variation of **documentProc**.

MacExtras uses to create the About Program window (see Listing 4.3, Lines 349–350), this window record is on the heap because the first parameter to **NewWindow** is **NIL** (148). The window manager understands this as an instruction to reserve memory on the heap for storing various information it needs to keep track of a program's windows.

As long as that works—it didn't if **wPtr** is **NIL** at line 150—the remaining statements save the window pointer in **wPtrs** (153), count the total number of windows now open (154), and display the result (155). Lines 156–158 adjust the appropriate menus, activating the Close command now that at least one window is open, and deactivating New if the number of windows equals the program's maximum. Line 152 has comments around it because it is unnecessary. This line changes the window title, already done earlier in the call to **NewWindow** (148–149). If you were loading a resource template instead of creating window records this way, you would enable line 152 to name the window before displaying it.

As you can see from this, the program does not use handles to refer to window records on the heap. You are correct in thinking that this breaks the rules of memory management at the start of this chapter but, in the case of window records, you cannot use handles to address them. Window records must be in non-relocatable memory blocks for “historical” reasons—meaning that was the way they were designed in the Mac's infant days. Because of this, there's always the risk that the heap will become fragmented with multiple window records.

This will not happen if you are careful to design handles to all other dynamic

objects your program uses and if you never lock those objects in the heap. When **NewWindow** creates new window records, the Memory Manager attempts to locate them as low as possible, moving relocatable blocks upwards to gain room. Therefore, even though you might fragment the heap with window records, the worst consequence usually is several holes of window record size near the heap bottom. Because the Memory Manager uses those holes for subsequent windows, the problem is not as severe as it otherwise might be.

Procedures **CloseProgramWindow** and **DoClose** (165–191) are similar to those in Apshell. The difference here is the addition of array index **wIndex**, which selects one out of many windows that might be open. As in the basic shell, **DoClose** assumes that if a window does not belong to the program, it must belong to a desk accessory. In that case it calls **CloseDAWindow** (190).

The only new element in **Initialize** (197–212) is the **FOR** loop at lines 209–210, which initializes all window pointers to **NIL**. This follows the general rule that pointer variables should be **NIL** if they don't point to actual objects. This is your responsibility. Pascal does not initialize pointers automatically.

QuitConfirmed (215–226) is subtly different from the same function in ApShell. The Program Engine calls it once at every program cycle (283), checking whether **quitRequested** is **TRUE** (222). If so, then the **WHILE** loop (223–224) examines **FrontWindow**. A **NIL** value indicates that no windows exist. As long as this condition is *not* met, the loop calls **DoClose** (224). Because of our earlier design rule that a close should work for both desk accessories and common windows, these actions totally clean up the desktop, removing all windows before the program ends, no matter to whom they belong. To see this work, open several windows and desk accessories and quit the program. The windows peel off one by one as the **WHILE** loop at line 223 cycles.

The rest of the program also is similar to ApShell. Instead of comparing **FrontWindow** with a global program variable, though, **DoSystemTask** calls **WindowIsOurs** (251) to check whether the frontmost window belongs to the program. If not, it enables the appropriate menu commands for active desk accessories.

TEXT IN WINDOWS

Up to now, most of our windows have been empty. It's time to put something in them, as the next example demonstrates. The program, MacStat, is a useful utility that displays several facts about your computer. Type in Listing 5.3, save as MACSTAT.R, and compile with RMaker. Type in Listing 5.4 and save as MACSTAT.PAS. Insert the appropriate lines from ApShell at lines 18 and 289 and compile with Turbo.

Running MacStat displays several facts about your system. It describes your computer model, the ROM version, the amount of memory you have, and several other items. In addition to using the program as is, you can extract its programming when your own projects need these same facts.

Listing 5.3. MACSTAT.R

```

1: *-----*
2: * MacStat.PAS resources -- Compile with RMaker      *
3: *-----*
4:
5: Programs:Windows.F:MacStat.RSRC          ;; Send output to here
6:
7:
8: *-----*
9: * About box string list                          *
10: *-----*
11: TYPE STR#                                ;; String list resource
12:     ,1 (32)                            ;; ID and attribute (purgeable)
13: 6                                       ;; Number of strings that follow
14: MacStat                                ;; Program name
15: by Tom Swan                          ;; Author
16: Version 1.00                        ;; Version number
17: (C) 1987 by Swan Software            ;; Copyright notice
18: P. O. Box 206, Lititz, PA 17543    ;; Address
19: (717)-627-1911                      ;; Telephone
20:
21:
22: *-----*
23: * The Apple Info menu                          *
24: *-----*
25:
26: TYPE MENU
27:     ,1                                ;; Menu ID number to use in program
28:     \14                              ;; Bitten-apple graphics symbol
29:     About MacStat...                ;; The command as shown in menu
30:     (-                             ;; Divider line between command and desk accs.
31:
32:
33: *-----*
34: * The File menu                              *
35: *-----*
36:
37: TYPE MENU
38:     ,2                                ;; Menu ID number to use in program
39: File                                ;; Menu title as shown in menu bar
40:     New /N
41:     (Close
42:     (-
43:     Quit /Q
44:
45:
46: *-----*
47: * The Edit menu                              *
48: *-----*
49:
50: TYPE MENU
51:     ,3
52: Edit
53:     (Undo /Z
54:     (-
55:     (Cut /X
56:     (Copy /C
57:     (Paste /V
58:     (Clear
59:
60:
61: *-----*
62: * Window template                          *
63: *-----*
64:

```

```

65: TYPE WIND
66:     ,1 (32)                ;; ID number and attribute (purgeable)
67: MacStat                   ;; Window title
68: 46 7 328 502              ;; top, left, bottom, right coordinates
69: Visible GoAway            ;; Visible window with close button
70: 8                          ;; Standard doc window with grow & zoom boxes
71: 0                          ;; Window reference (none)
72:
73:
74:
75: * END

```

Listing 5.4. MACSTAT.PAS

```

1: {$O Programs:Windows.F: }           { Send compiled code to here }
2: {$R Programs:Windows.F:MacStat.Rsrc} { Use this compiled resource file }
3: {$U-}                               { Turn off standard library units }
4:
5:
6: PROGRAM MacStat;
7:
8: (*
9:
10:  * PURPOSE : Display various facts about your Mac
11:  * SYSTEM  : Macintosh / Turbo Pascal
12:  * AUTHOR   : Tom Swan
13:
14:  *)
15:
16:
17:
18: << INSERT LINES 17-203 FROM APSHELL.PAS >>
19:
20:
21:
22: PROCEDURE DrawContents( whichWindow : WindowPtr );
23:
24: { Display statistics in whichWindow. }
25:
26:     CONST
27:
28:         HWCfgFlags      = $B22;          { Hardware configuration flags }
29:
30:     TYPE
31:
32:         WordPointer     = ^INTEGER;      { Pointer to integer }
33:
34:     VAR
35:
36:         rom, machine    : INTEGER;        { System identifiers }
37:         secs             : LONGINT;        { Time/date encoded in seconds }
38:         sp              : SysPPtr;        { Pointer to system information }
39:         textWidth       : INTEGER;        { Maximum width of a character }
40:         textHeight      : INTEGER;        { Height of a character }
41:         flags            : WordPointer;    { Pointer to low memory flags }
42:         oldClipRgn      : RgnHandle;      { Handle to old clipping region }
43:
44:

```

(continued)

```

45:  PROCEDURE DrawAt( x, y : INTEGER; s : Str255 );
46:
47:  { Draw string at this coordinate, similar to an x,y location }
48:  { on a conventional computer terminal. }
49:
50:      CONST
51:
52:          Xoffset = 5;      { Blank pixels in left border }
53:          Yoffset = 12;     { Blank pixels in top border }
54:
55:      BEGIN
56:          MoveTo( Xoffset + x * textWidth , Yoffset + y * textHeight );
57:          DrawString( s )
58:      END; { DrawAt }
59:
60:
61:  PROCEDURE NewFont( fontNumber, pointSize : INTEGER );
62:
63:  { Select new font and adjust variables for DrawAt }
64:
65:
66:      BEGIN
67:          TextFont( fontNumber );      { Use new font }
68:          TextSize( pointSize );      { Use new size }
69:          textHeight := pointSize + 2; { Calc new character height }
70:          textWidth := CharWidth( 'M' ) { Calc width of widest char }
71:      END; { NewFont }
72:
73:
74:  PROCEDURE ShowPort( port : INTEGER );
75:
76:  { Display serial port configuration encoded as an integer }
77:
78:      VAR
79:
80:          baudRate, dataBits, stopBits, parity : INTEGER;
81:
82:      BEGIN
83:          baudRate := BitAnd( port, $3FF );
84:          dataBits := BitAnd( port, $C00 );
85:          parity   := BitAnd( port, $3000 );
86:          stopBits := BitAnd( port, $C000 );
87:
88:          CASE baudRate OF
89:              baud300    : DrawString( '3' );
90:              baud600    : DrawString( '6' );
91:              baud1200   : DrawString( '12' );
92:              baud2400   : DrawString( '24' );
93:              baud3600   : DrawString( '36' );
94:              baud4800   : DrawString( '48' );
95:              baud7200   : DrawString( '72' );
96:              baud9600   : DrawString( '96' );
97:              baud19200  : DrawString( '192' );
98:              baud57600  : DrawString( '576' );
99:          OTHERWISE
100:             DrawString( '??' );
101:          END; { case }
102:          DrawString( '00 baud / ' );
103:
104:          CASE dataBits OF
105:              data5      : DrawString( '5' );
106:              data6      : DrawString( '6' );
107:              data7      : DrawString( '7' );
108:              data8      : DrawString( '8' );
109:          OTHERWISE
110:             DrawString( '??' );
111:          END; { case }
112:          DrawString( ' data bits / ' );
113:

```

```

114:         CASE stopBits OF
115:             stop10      : DrawString( '1' );
116:             stop15      : DrawString( '1.5' );
117:             stop20      : DrawString( '2' );
118:         OTHERWISE
119:             DrawString( '??' );
120:         END; { case }
121:         DrawString( ' stop bits / ' );
122:
123:         CASE parity OF
124:             noParity     : DrawString( 'no' );
125:             oddParity    : DrawString( 'odd' );
126:             evenParity   : DrawString( 'even' );
127:         OTHERWISE
128:             DrawString( 'no' ); { catch the 2nd of 2 no-parity settings }
129:         END; { case }
130:         DrawString( ' parity' )
131:
132:     END; { ShowPort }
133:
134:
135:     PROCEDURE ShowTime( secs : LONGINT );
136:
137:     { Display date & time encoded as no. of secs from 1/1/04 }
138:
139:     VAR
140:
141:         s : Str255;
142:
143:     BEGIN
144:         IUDateString( secs, longDate, s );
145:         DrawString( s );
146:         DrawString( ' (' );
147:         IUTimeString( secs, TRUE, s );
148:         DrawString( s );
149:         DrawString( ')' );
150:     END; { ShowTime }
151:
152:
153:     PROCEDURE DrawInteger( n : LONGINT );
154:
155:     { Display integer value }
156:
157:     VAR
158:
159:         s : Str255;
160:
161:     BEGIN
162:         NumToString( n, s ); { Convert number to string }
163:         DrawString( s )      { Display it }
164:     END; { DrawInteger }
165:
166:
167:
168:     PROCEDURE ShowFontName( fn : INTEGER );
169:
170:     { Display name of font fn or its number if unknown }
171:
172:     VAR
173:
174:         fontName : Str255;
175:
176:
177:     BEGIN
178:
179:         GetFontName( fn, fontName );
180:         DrawString( fontName )
181:
182:     END; { ShowFontName }

```

(continued)


```

183:
184:
185:
186:   PROCEDURE DrawStats;
187:
188:   { Display information in window }
189:
190:   BEGIN
191:     Environs( rom, machine );      { Get system environment info }
192:
193:     NewFont( applFont, 12 );      { Use application font, 12 points }
194:     TextFace( [ bold, underline, italic ] ); { in this text style }
195:     DrawAt( 3, 2, 'About your Mac...' );
196:
197:     NewFont( monaco, 9 );          { Use monaco font, 9 points }
198:     TextFace( [ ] );               { in plain style text }
199:
200:     DrawAt( 5, 5, 'Model ..... Macintosh ' );
201:     IF machine = macXLMachine
202:     THEN
203:       DrawString( 'XL' )
204:     ELSE
205:       BEGIN { Distinguish among 128K / 512K / and Mac Plus }
206:         flags := POINTER( HWCfgFlags ); { Assign ptr address }
207:         IF flags^ < 0
208:         THEN DrawString( 'Plus' ) { Mac Plus if bit 15=1 }
209:         ELSE IF LONGINT( TopMem ) > 500000 { else guess }
210:         THEN DrawString( '512K' )
211:         ELSE DrawString( '128K' )
212:       END; { else }
213:
214:       DrawAt( 5, 7, 'ROM Version ... ' );
215:       DrawInteger( rom );
216:
217:
218:       DrawAt( 5, 8, 'ROM Size ..... ' );
219:       IF rom >= 117
220:       THEN DrawString( '128K' )
221:       ELSE DrawString( '64K' );
222:
223:       DrawAt( 5, 10, 'Total memory .. ' );
224:       DrawInteger( LONGINT( TopMem ) );
225:       DrawString( ' bytes' );
226:
227:       GetDateTime( secs );
228:       DrawAt( 5, 12, 'Date & time ... ' );
229:       ShowTime( secs );
230:
231:       sp := GetSysPPtr;
232:       WITH sp^ DO
233:       BEGIN
234:         DrawAt( 5, 13, 'Alarm set .... ' );
235:         ShowTime( alarm );
236:         DrawAt( 5, 15, 'Default font .. ' );
237:         ShowFontName( font+1 );
238:         DrawAt( 5, 17, 'PortA config .. ' );
239:         ShowPort( portA );
240:         DrawAt( 5, 18, 'PortB config .. ' );
241:         ShowPort( portB );
242:         DrawAt( 5, 20, 'Printer port .. Port ' );
243:         IF ODD( kbdPrint )
244:         THEN DrawChar( 'B' )
245:         ELSE DrawChar( 'A' )
246:       END { with }
247:
248:     END; { DrawStats }
249:
250:

```

```

251:  PROCEDURE ClipToViewArea;
252:
253:  { Set clipping area to window contents inside scroll bars }
254:
255:  VAR
256:
257:      r : Rect;
258:
259:  BEGIN
260:
261:      oldClipRgn := NewRgn;           { Create region for saving }
262:      GetClip( oldClipRgn );          { at oldClipRgn handle }
263:
264:      r := whichWindow^.portRect;     { Copy window's portRect }
265:      WITH r DO
266:  BEGIN
267:          right := right - scBarWidth; { Reduce rect to exclude }
268:          bottom := bottom - scBarWidth { scroll bars }
269:      END; { with }
270:      ClipRect( r )                   { Clip to this new area }
271:
272:  END; { ClipToViewArea }
273:
274:
275:  BEGIN { DrawContents }
276:
277:      EraseRect( whichWindow^.portRect ); { Standard update }
278:      DrawScrollBars( whichWindow );      { actions }
279:
280:      ClipToViewArea;                     { Change clipping and save old region }
281:      DrawStats;                           { Display information in window }
282:      SetClip( oldClipRgn );               { Restore old clipping region }
283:      DisposeRgn( oldClipRgn )            { Dispose handle to free heap space }
284:
285:  END; { DrawContents }
286:
287:
288:
289: << INSERT LINES 220-456 FROM APSHELL.PAS >>
290:
291:
292:
293: { END MacStat }

```

MacStat Play-by-Play

MACSTAT.PAS (1-293)

Because there's nothing special about this program's resource text file, we'll skip straight to the main listing. Line 28 assigns the address of the computer's hardware configuration flags to constant **HWCfgFlags**, an address that Apple Computer guarantees will never change. Later on, line 206 sets a pointer to this address, allowing the program to examine the flags. In this case, all we want is to check bit 15. If this bit is 1, then the computer has a Macintosh Plus logic board.

Procedure **DrawAt** (45-58) is a useful tool you can extract for your own programs. It takes three parameters. Integers **x** and **y** specify a coordinate in the window where you want to display the third parameter, string **s**. That coordinate is

not quite the same as on a conventional terminal or in Turbo Pascal's textbook interface, although **DrawAt** attempts to align columns as best it can.

NewFont (61–71) is another tool you can use. It takes a font number and size, changing the current window's text font to those parameters. Also, the procedure sets **textHeight** and **textWidth**, which **DrawAt** uses to calculate where to display text. Notice that line 70 sets **textWidth** to the pixel-width of a capital M. It does this because M usually is the widest character in a font.

ShowPort (74–132) is a long procedure that displays the settings of the two serial ports. Parameter **port** holds the encoded information about the port's settings obtained at line 231. Although you may never need to display the port settings, you can use lines 83–86 to decode the baud rate (meaning bits per second) and other port settings as shown here.

Procedure **ShowTime** (135–150) demonstrates how to convert the date and time, stored as the number of seconds from midnight, January 1, 1904. Lines 144–145 extract and display the date. The remaining lines extract and display the time. Change **longDate** (144) to either **shortDate** or to **abbrevDate** to display different formats. Change **TRUE** to **FALSE** in **IUTimeString** (147) if you don't want to display seconds.

DrawInteger (153–164) is a simple tool that displays value **n** at the current pen location. It first converts the number into a string (162) and then displays that string. **ShowFontName** (168–182) calls **GetFontName** (179) to set string variable **fontName** to the title of the font with number **fn**. It then displays that string.

DrawStats to END (186–293)

DrawStats displays all of the statistics you see in MacStat's window. Line 191 calls **Environs** to retrieve certain system information, setting parameter **rom** to the ROM version number, and **machine** to either **macXLMachine** if this is a Macintosh XL, or to **macMachine** if not. The procedure uses this value along with **HWCfgFlags** bit 15 to determine what computer the program is running (200–212). You should be able to understand the rest of the procedure with no further explanation.

An important tool for which you'll find a variety of uses is procedure **ClipToViewArea** (251–272). The procedure prevents the error that Figure 5.8 illustrates. After shrinking the window, you notice that text overshoots the scroll bar boundaries to the right and bottom. Obviously this is wrong.

The problem occurs because the window manager initially sets clipping—the area to where it limits visible drawing—to the entire coordinate plane, which as you recall, is 65,535 points square. When you draw something in a window, the system restricts what you see to the combination of the clipping area (**clipRgn**), the window boundary (**portRect**), and the visible area (**visRgn**). But that does not account for the scroll bar outlines and, therefore, drawing overwrites them.

To fix the problem, **ClipToViewArea** excludes the scroll bar rectangles from the window's clipping region. It first saves the current region in two steps (261–262).

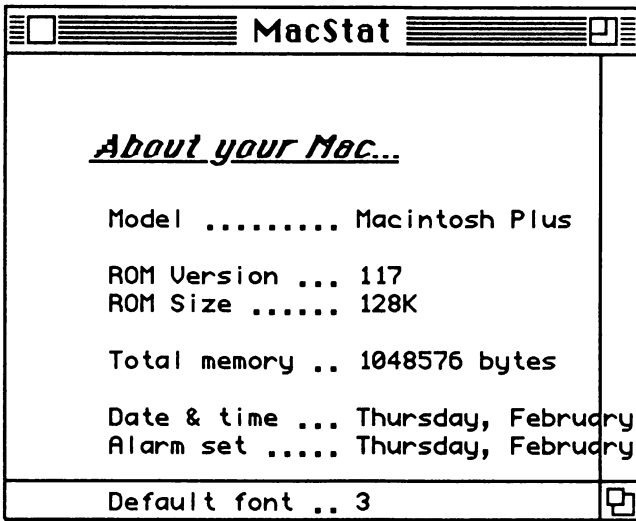


Figure 5.8 Oops! Displaying text in windows without clipping to the scroll bar outlines can produce the error illustrated here.

NewRgn starts a new region structure on the heap, passing back a handle, which the program stores in **oldClipRgn**. Regions are relocatable memory blocks of varying size. (You don't need to be concerned with their in-memory structure.) Procedure **GetClip** then assigns the current clipping region to the initialized handle. When **DrawContents** is done, it reverses these steps, calling **SetClip** (282) to restore the original clipping region and **DisposeRgn** (283) to dispose of the handle and its associated memory block on the heap. Always remember to dispose your handles when you're done using them.

ClipToViewArea sets the new clipping region to rectangle **r**. It first calculates the rectangle by setting it equal to the window's **portRect** (264) and subtracting the scroll bar width from the **right** and **bottom** fields (267–268). Then it passes this rectangle to **ClipRect**, which restricts drawing to the new area, protecting the scroll bars from overwriting.

PICTURE WINDOWS

To display graphics, you can set up windows to operate like tape recorders, saving drawing commands you give and then executing those commands later as needed. This lets you draw complex objects in windows and let the window manager update your drawing rather than repeating those same commands in your update handler procedure. For example, when you move one window aside to uncover

another underneath, the window manager automatically redraws the uncovered portion of your drawing—you don't have to respond to the event yourself.

The next example explains how to do this. Type in Listing 5.5, save as PICTURE.R, and compile with RMaker. Type in Listing 5.6, save as PICTURE.PAS, and compile with Turbo. Notice that lines 48, 90, 93, 116, and 131 tell you to insert various lines from ApShell. When you run the program, choose the File menu's New command to display a window with a picture inside. Open a desk accessory or two (Note Pad is a good choice), drag the windows, and move them on top of each other. Notice that the window manager always redraws only as much of the picture as it needs. This happens automatically without responding to update events for this window as in previous examples.

Listing 5.5. PICTURE.R

```

1: *-----*
2: * Picture.PAS resources -- Compile with RMaker      *
3: *-----*
4:
5: Programs:Windows.F:Picture.RSRC    ;; Send output to here
6:
7:
8: *-----*
9: * About box string list                *
10: *-----*
11:
12: TYPE STR#                                ;; String list resource
13:   ,1 (32)                                ;; ID and attribute (purgeable)
14:   6                                       ;; Number of strings that follow
15: Picture Window                          ;; Program name
16: by Tom Swan                            ;; Author
17: Version 1.00                           ;; Version number
18: (C) 1987 by Swan Software               ;; Copyright notice
19: P. O. Box 206, Lititz, PA 17543        ;; Address
20: (717)-627-1911                         ;; Telephone
21:
22:
23: *-----*
24: * The Apple Info menu                  *
25: *-----*
26:
27: TYPE MENU
28:   ,1                                     ;; Menu ID number to use in program
29:   \14                                   ;; Bitten-apple graphics symbol
30:   About Picture...                     ;; The command as shown in menu
31:   (-)                                  ;; Divider line between command and DAs
32:
33:
34: *-----*
35: * The File menu                        *
36: *-----*
37:
38: TYPE MENU
39:   ,2                                     ;; Menu ID number to use in program
40: File                                   ;; Menu title as shown in menu bar
41:   New /N
42:   (Close
43:   (-
44:   Quit /Q
45:
46:

```

```

47: *-----*
48: * The Edit menu *
49: *-----*
50:
51: TYPE MENU
52:   ,3
53: Edit
54:   (Undo /Z
55:   (-
56:   (Cut /X
57:   (Copy /C
58:   (Paste /V
59:   (Clear
60:
61:
62: *-----*
63: * Window template *
64: *-----*
65:
66: TYPE WIND
67:   ,1 (32)           ;; ID number and attribute (purgeable)
68:   Picture Window    ;; Window title
69:   45 10 200 300      ;; top, left, bottom, right coordinates
70:   Visible GoAway     ;; Visible window with close button
71:   4                  ;; Standard window without scroll bars
72:   0                  ;; Window reference (none)
73:
74:
75:
76: * END

```

Listing 5.6. PICTURE.PAS

```

1: {$O Programs:Windows.F: }           { Send compiled code to here }
2: {$R Programs:Windows.F:Picture.Rsrc} { Use this compiled resource file }
3: {$U-}                               { Turn off standard library units }
4:
5:
6: PROGRAM Picture;
7:
8: (*
9:
10:  * PURPOSE : Demonstrate QuickDraw Pictures in Windows
11:  * SYSTEM  : Macintosh / Turbo Pascal
12:  * AUTHOR  : Tom Swan
13:
14:  *)
15:
16:
17: {$U Programs:Units.F:MacExtras }      { Open this library unit file }
18:
19:
20:   USES
21:
22:     Memtypes, QuickDraw, OSIntf, ToolIntf, PackIntf, MacExtras;
23:
24:
25:
26:   CONST
27:

```

(continued)

```

28:      FileID      = 2;      { File menu Resource ID and commands }
29:      NewCmd       = 1;
30:      CloseCmd     = 2;
31:      {-----}
32:      QuitCmd      = 4;
33:
34:      WindowID     = 1;      { Window resource ID }
35:
36:
37:
38:      VAR
39:
40:      wRec          : WindowRecord;      { Program's window data record }
41:      wPtr          : WindowPtr;         { Pointer to above wRec }
42:
43:      quitRequested : BOOLEAN;           { TRUE if quitting }
44:      windowOpen    : BOOLEAN;           { TRUE only if window is open }
45:
46:
47:
48: << INSERT LINES 61-74 FROM APSHELL.PAS >>
49:
50:
51: PROCEDURE DoNew;
52:
53: { Open window and attach QuickDraw picture }
54:
55:      VAR
56:
57:      ph : PicHandle;
58:      r : Rect;
59:      i : INTEGER;
60:
61:      BEGIN
62:          IF NOT windowOpen THEN
63:              BEGIN
64:                  wPtr := GetNewWindow( WindowID, @Wrec, POINTER( -1 ) );
65:                  windowOpen := wPtr <> NIL;
66:                  IF windowOpen THEN
67:                      BEGIN
68:                          SetPort( wPtr );
69:                          ClipRect( wPtr^.portRect ); { Not optional! }
70:
71:                          r := wPtr^.portRect; { Prepare rectangle }
72:                          InsetRect( r, 25, 25 );
73:                          ph := OpenPicture( r ); { Create a picture }
74:                          InsetRect( r, 10, 10 );
75:                          FOR i := 1 TO 20 DO { Draw into picture }
76:                              BEGIN
77:                                  FrameOval( r );
78:                                  InsetRect( r, 4, 0 )
79:                              END; { for }
80:                          ClosePicture; { Close picture }
81:                          SetWindowPic( wPtr, ph ); { Attach to window }
82:
83:                          EnableItem( fileMenu, CloseCmd );
84:                          DisableItem( fileMenu, NewCmd )
85:                      END
86:                  END { if }
87:              END; { DoNew }
88:
89:
90: << INSERT LINES 96-175 FROM APSHELL.PAS >>
91:
92:
93: << INSERT LINES 227-298 FROM APSHELL.PAS >>
94:
95:

```

```

96: PROCEDURE ActivateEvents;
97:
98: { Activate or deactivate windows }
99:
100: BEGIN
101:     WITH theEvent DO
102:     BEGIN
103:
104:         whichWindow := WindowPtr( message ); { Extract window pointer }
105:         SetPort( whichWindow ); { Change current port }
106:
107:         IF BitAnd( modifiers, activeFlag ) <> 0
108:         THEN FixEditMenu( FALSE ) { Activate a window }
109:         ELSE FixEditMenu( TRUE ) { Deactivate a window }
110:
111:     END { with }
112: END; { ActivateEvents }
113:
114:
115:
116: << INSERT LINES 350-368 FROM APSHELL.PAS >>
117:
118:
119: PROCEDURE Initialize;
120:
121: { Program calls this routine one time at start }
122:
123: BEGIN
124:     SetUpMenuBar; { Initialize and display menus }
125:     quitRequested := FALSE; { TRUE on selecting Quit command }
126:     windowOpen := FALSE; { TRUE after using New command }
127:     DisplayAboutBox { Identify program }
128: END; { Initialize }
129:
130:
131: << INSERT LINES 382-393 FROM APSHELL.PAS >>
132:
133:
134: PROCEDURE DoSystemTasks;
135:
136: { Do operations at each pass through main program loop }
137:
138: BEGIN
139:
140:     SystemTask; { Give DAs their fair share of time }
141:
142:     IF FrontWindow = NIL THEN
143:
144:         BEGIN { Set up menu commands for empty desktop }
145:
146:             FixEditMenu( FALSE );
147:             EnableItem( fileMenu, NewCmd );
148:             DisableItem( fileMenu, CloseCmd );
149:
150:         END ELSE
151:
152:         IF FrontWindow <> wPtr THEN
153:
154:             BEGIN { Set up menu commands for active desk accessory }
155:
156:                 FixEditMenu( TRUE );
157:                 EnableItem( fileMenu, CloseCmd );
158:
159:             END { else / if }
160:
161:         END; { DoSystemTasks }
162:
163:

```

(continued)


```

164: BEGIN
165:
166:   Initialize;
167:
168:   REPEAT
169:
170:     DoSystemTasks;
171:
172:     IF GetNextEvent( everyEvent, theEvent ) THEN
173:
174:       CASE theEvent.what OF
175:
176:         MouseDown   : MouseDownEvents;
177:         KeyDown     : KeyDownEvents;
178:         AutoKey      : { ignored };
179:         ActivateEvt  : ActivateEvents
180:
181:       END { case }
182:
183:   UNTIL QuitConfirmed
184:
185: END.

```

Picture Play-by-Play

PICTURE.PAS (1-185)

Picture's global declarations are similar to previous examples. Line 40 declares a window record as a global variable instead of on the heap as in *MultiWind*. Although this permanently takes space in the application's global memory area, it's an acceptable method for small examples, especially those that have only a single window.

Procedure **DoNew** (51-87) demonstrates how to open a window and attach a picture. Line 57 declares a picture handle (**PicHandle**) that the window manager uses to store drawing commands in a relocatable memory block on the heap. After loading the resource template and displaying the window (64-65), the program prepares the window for attaching a picture, setting the current port to the program window (68), and reducing the clipping region to inside its boundaries (69). These steps are required when attaching pictures to windows—don't forget them.

Having prepared the window, lines 71-72 initialize a rectangle to the size of the picture the procedure draws. Line 73 passes this rectangle to **OpenPicture**, which starts a new picture on the heap and passes back a handle to the associated memory block. With the picture open, all QuickDraw commands divert to the picture instead of the screen. The window manager traps the drawing commands (74-79), storing them in memory in a form it can later use during window update events.

After finishing drawing, line 80 calls **ClosePicture** so that future QuickDraw commands again go to the screen. The final step is in line 81, where **SetWindowPic** attaches the picture handle **ph** to the window. From then on, the window manager is able to automatically display the picture window with no further help from the program. Except for a few changes, the rest of the program is similar to *ApShell*.

TEXT AND SCROLL BARS

The final programs in this chapter develop a set of tools for displaying text in windows and adding horizontal and vertical scroll bars. The tools operate similarly to the Turbo editor, but they do not allow you to change text or type anything. (The next chapter explains ways to enter information in programs.) You might use these tools to include on-line instructions on program disks or to display help screens.

There are three listings in the set. Listing 5.7 is a unit that contains tools you can add to any program. Type it in and save as TEXTUNIT.PAS. Compile with Turbo to a disk code file. Before compiling, you previously must have typed in and compiled the MacExtras unit from Chapter 4. An explanation of TextUnit's tools ends this chapter.

Listing 5.8 is the resource text file that goes with the program. Type it in, save as READER.R, and compile with RMaker. Then type in Listing 5.9, save as READER.PAS, and compile. The program, Reader, demonstrates how to use the TextUnit tools. It reads file ReadMe, which traditionally contains last minute notes about programs on disk.

Reader simply ends if it does not find a ReadMe file on the same volume and folder from which you run it. A better program would of course display an error message or let you view other files—but those are subjects for the next chapter. To test the program, use the Turbo editor to create a ReadMe file. Or rename any MacWrite file saved with the text-only option. Specify carriage returns at the ends of paragraphs if you want TextUnit to automatically adjust lines to fit inside the window borders.

Listing 5.7. TEXTUNIT.PAS

```

1: {$O Programs:Units.F: }           { Send compiled code to here }
2: {$U-}                             { Turn off standard library units }
3:
4:
5: UNIT TextUnit( 132 );
6:
7: (*
8:
9:  * PURPOSE : Text display tools
10:  * SYSTEM  : Macintosh / Turbo Pascal
11:  * AUTHOR   : Tom Swan
12:
13: *)
14:
15:
16: INTERFACE                         { Items visible to a host program }
17:
18:
19: {$U Programs:Units.F:MacExtras }   { Open this library unit file }
20:
21:

```

(continued)

```

22:  USES
23:
24:      Memtypes, QuickDraw, OSIntf, ToolIntf, PackIntf, MacExtras;
25:
26:
27:  TYPE
28:
29:      TUHandle = ^TUPtr;    { Pointer to a TUREc pointer }
30:      TUPtr = ^TUREc;      { Pointer to a TUREc variable }
31:
32:      TUREc =
33:          RECORD
34:              textRech      : TEHandle;      { Handle to TextEdit record }
35:              hBarH         : ControlHandle;  { Handle to horiz scroll bar }
36:              vBarH         : ControlHandle;  { Handle to vert scroll bar }
37:              linesPerPage  : INTEGER;        { Max lines in window height }
38:              textWidth     : INTEGER         { Maximum width in pixels }
39:          END; { TUREc }
40:
41:
42:
43:  FUNCTION TUAttach( wPtr      : WindowPtr;
44:                    paragraphs : BOOLEAN;
45:                    columnInches : INTEGER      ) : BOOLEAN;
46:
47:  PROCEDURE TUDispose( wPtr : WindowPtr );
48:
49:  PROCEDURE TUClick( wPtr : WindowPtr; where : Point );
50:
51:  PROCEDURE TUUpdate( wPtr : WindowPtr );
52:
53:  PROCEDURE TUAActivate( wPtr : WindowPtr; activate : BOOLEAN );
54:
55:  PROCEDURE TUREsize( wPtr : WindowPtr );
56:
57:  FUNCTION TUReadText( wPtr      : WindowPtr;
58:                     fileName   : Str255;
59:                     volNum     : INTEGER      ) : BOOLEAN;
60:
61:
62:
63:  IMPLEMENTATION      { Items not visible to a host program }
64:
65:
66:  CONST
67:
68:      overlap      = 4;      { Lines to overlap when paging up and down }
69:      blankMargin  = 4;      { Text window margins in pixels }
70:      colWidth     = 16;     { Minimum horiz scroll amount in pixels }
71:      oneInch      = 72;     { Number of pixels in 1 inch (horiz) }
72:
73:
74:  VAR
75:
76:      theTUHand    : TUHandle;    { Currently active TUREc handle }
77:
78:
79:
80:  FUNCTION GetTUHandle( wPtr : WindowPtr ) : BOOLEAN;
81:
82:  { Returns TRUE if it can extract and verify global theTUHand from wPtr
83:    refCon field. }
84:
85:  { LOCAL TO UNIT }
86:

```

```

87: BEGIN
88:   theTUHand := TUHandle( WindowPeek( wPtr )^.refCon );
89:   GetTUHandle := theTUHand <> NIL
90: END; { GetTUHandle }
91:
92:
93: PROCEDURE ResizeScrollBar( wPtr      : WindowPtr;
94:                             barType   : CHAR;
95:                             cHand     : ControlHandle );
96:
97: { Resize and move scroll bar control addressed by cHand in window at wPtr.
98:   BarType should be 'H' for horizontal or 'V' for vertical scroll bars.
99:   NOTE: This procedure makes the control invisible. Call ShowControl
100:  after. The reason for this requirement is to let other procedures set
101:  control values before finally displaying a relocated scroll bar. }
102:
103: { LOCAL TO UNIT }
104:
105:   VAR
106:
107:     wHeight : INTEGER; { Full window height in pixels }
108:     wWidth  : INTEGER; { Full window width in pixels }
109:     ch      : INTEGER; { Control h local coordinate value }
110:     cv      : INTEGER; { Control v local coordinate value }
111:     cWidth  : INTEGER; { Control width in pixels }
112:     cHeight : INTEGER; { Control height in pixels }
113:
114: BEGIN
115:   HideControl( cHand ); { Make invisible, if it's not already. }
116:   WITH wPtr^.portRect DO { Resize using window's local coordinates. }
117:   BEGIN
118:     wHeight := 2 + ( bottom - top ); { Calculate window height }
119:     wWidth  := 2 + ( right - left ); { and width in pixels. }
120:
121:     IF barType = 'H' THEN
122:       BEGIN { Calculate horizontal scroll bar sizes }
123:         ch := -1;
124:         cv := bottom - ScBarWidth;
125:         cHeight := ScBarWidth + 1;
126:         cWidth := wWidth - ScBarWidth
127:       END ELSE
128:       BEGIN { Calculate vertical scroll bar sizes }
129:         ch := right - ScBarWidth;
130:         cv := -1;
131:         cHeight := wHeight - ScBarWidth;
132:         cWidth := ScBarWidth + 1
133:       END; { else }
134:
135:       MoveControl( cHand, ch, cv ); { Move to new location }
136:       SizeControl( cHand, cWidth, cHeight ) { Change to new size }
137:
138:     END; { with }
139:     ValidRect( cHand^.ctrlRect ) { Remove control from update region }
140:   END; { ResizeScrollBar }
141:
142:
143: PROCEDURE MakeNewScroll( wPtr      : WindowPtr;
144:                           barType   : CHAR;
145:                           VAR barHandle : ControlHandle );
146:
147: { Create a new scroll bar, attach it to window at wPtr, and return handle
148:  to the control in barHandle. BarType should be 'V' for vertical bars,
149:  or 'H' for horizontal bars. Any errors return barHandle = NIL. }
150:
151: { LOCAL TO UNIT }
152:

```

(continued)

```

153: BEGIN
154:
155:     barHandle :=
156:         NewControl( wPtr,           { Attach control to this window }
157:                     wPtr^.portRect, { Any Rect will do--resized later }
158:                     '',              { Null string (no title needed) }
159:                     FALSE,           { Make temporarily invisible }
160:                     0, 0, 0,          { Initialize value, min, and max }
161:                     scrollBarProc,   { The scroll bar procedure id }
162:                     0 );             { Reference value (none) }
163:
164:     IF barHandle <> NIL THEN
165:         BEGIN
166:             ResizeScrollBar( wPtr, barType, barHandle );
167:
168:         END { if }
169:
170:     END; { MakeNewScroll }
171:
172:
173: PROCEDURE CheckNoScroll;
174:
175: { Test control values. If either is zero, test whether control should
176:   be enabled. If not, disable by setting control max to zero. }
177:
178: VAR
179:
180:     vWidth, dWidth : INTEGER;
181:     ch, cv : ControlHandle;
182:     lines, lpp : INTEGER;
183:
184: BEGIN
185:
186:     WITH theTUHand^^, textRecH^^ DO
187:         BEGIN
188:             ch := hBarH;
189:             cv := vBarH;
190:             vWidth := viewRect.right - viewRect.left;
191:             dWidth := destRect.right - destRect.left;
192:             lines := nLines;
193:             lpp := linesPerPage
194:         END; { with }
195:
196:         IF GetCtlValue( ch ) = 0 THEN
197:             IF vWidth >= dWidth
198:                 THEN SetCtlMax( ch, 0 );
199:
200:             IF GetCtlValue( cv ) = 0 THEN
201:                 IF lines <= lpp
202:                     THEN SetCtlMax( cv, 0 )
203:
204:         END; { CheckNoScroll }
205:
206:
207: PROCEDURE SetScBarValues( horizValue, vertValue : INTEGER );
208:
209: { Set scroll bar vertical and horizontal values and calculate min, and max
210:   settings based on TUREC parameters. Assumes theTUHand global handle is
211:   set properly. }
212:
213: { LOCAL TO UNIT }
214:
215: VAR
216:
217:     dWidth : INTEGER;
218:

```

```

219: BEGIN
220:
221:     HLock( Handle( theTUHand ) );
222:
223:     WITH theTUHand^^ DO
224:     BEGIN
225:
226:         HideControl( vBarH );
227:         SetCtlMin( vBarH, 0 );
228:         SetCtlMax( vBarH, textRecH^^.nLines );
229:         SetCtlValue( vBarH, vertValue );
230:         ShowControl( vBarH );
231:
232:         WITH textRecH^^ DO
233:             dWidth := destRect.right - destRect.left;
234:
235:         HideControl( hBarH );
236:         SetCtlMin( hBarH, 0 );
237:         SetCtlMax( hBarH, dWidth );
238:         SetCtlValue( hBarH, horizValue );
239:         ShowControl( hBarH );
240:
241:     END; { with }
242:
243:     HUnlock( Handle( theTUHand ) );
244:
245:     CheckNoScroll
246:
247: END; { SetScBarValues }
248:
249:
250: PROCEDURE PageText;
251:
252: { Display a new page of text according to the scroll bar position.
253:   Assumes theTUHandle global variable is set correctly. }
254:
255: { LOCAL TO UNIT }
256:
257: VAR
258:
259:     dh, dv : INTEGER; { Pixels to scroll horizontally, vertically }
260:
261: BEGIN
262:
263:     WITH theTUHand^^, textRecH^^ DO { HLock not needed here }
264:     BEGIN
265:
266:         dv :=
267:             ( viewRect.top - destRect.top ) - { Line offset }
268:             ( GetCtlValue( vBarH ) * lineHeight ); { Pixels to line }
269:
270:         dh :=
271:             ( viewRect.left - destRect.left ) - { Column offset }
272:             ( GetCtlValue( hBarH ) ); { Pixels to column }
273:
274:         IF ( dh <> 0 ) OR ( dv <> 0 )
275:             THEN TEScroll( dh, dv, textRecH )
276:
277:     END; { with }
278:
279:     (*CheckNoScroll*)
280:
281: END; { PageText }
282:
283:

```

(continued)

```

284: PROCEDURE ScrollText( theControl : ControlHandle; partCode : INTEGER );
285:
286: { Scroll text in window. Called as a TrackControl
287:   action procedure. For that reason, the parameter list must be exactly
288:   as declared here. Assumes that global theTUHandle is set correctly. }
289:
290: { LOCAL TO UNIT }
291:
292:
293:   VAR
294:
295:       ctlValue      : INTEGER;      { Current control value }
296:       ctlMax        : INTEGER;      { Control's maximum setting }
297:       ctlMin        : INTEGER;      { Control's minimum setting }
298:       lineColFull   : INTEGER;      { Amount to scroll for line or column }
299:       pageFull      : INTEGER;      { Amount to scroll for one page }
300:       amount        : INTEGER;      { Actual amount to scroll }
301:
302: BEGIN
303:
304:     ctlValue := GetCtlValue( theControl );
305:     ctlMax   := GetCtlMax( theControl );
306:     ctlMin   := GetCtlMin( theControl );
307:
308:     WITH theTUHand^^, textRech^^.viewRect DO
309:         IF theControl = vBarH THEN
310:             BEGIN { Vertical bars }
311:                 pageFull := linesPerPage - overlap; { Lines to page up/down }
312:                 lineColFull := 1 { Lines to scroll up/down }
313:             END ELSE
314:             BEGIN { Horizontal bars }
315:                 pageFull := ( right - left ) DIV 2; { Pixels to page lt/rt }
316:                 lineColFull := colWidth { Pixels to scroll lt/rt }
317:             END; { else }
318:
319:     amount := 0; { Prevents thumb from moving beyond ends }
320:
321:     CASE partCode OF
322:
323:         inUpButton :
324:             IF ctlValue > ctlMin
325:             THEN amount := -lineColFull; { Scroll DOWN or LEFT }
326:
327:         inDownButton :
328:             IF ctlValue < ctlMax
329:             THEN amount := lineColFull; { Scroll UP or RIGHT }
330:
331:         inPageUp :
332:             IF ctlValue > ctlMin
333:             THEN amount := -pageFull; { Page DOWN or LEFT }
334:
335:         inPageDown :
336:             IF ctlValue < ctlMax
337:             THEN amount := pageFull { Page UP or RIGHT }
338:
339:     END; { case }
340:
341:     IF amount <> 0 THEN { i.e. if not at end of control }
342:     BEGIN
343:         SetCtlValue( theControl, GetCtlValue( theControl ) + amount );
344:         PageText
345:     END { if }
346:
347: END; { ScrollText }
348:
349:

```

```

350: PROCEDURE FormatText( wPtr : WindowPtr; VAR dRect, vRect : Rect );
351:
352: { Calculates linesPerPage field of the TUREc associated with this window,
353:   and returns dest (format) and view (clipping) rectangles for TextEdit's
354:   TEREc. Assumes TUREc textWidth, vBarH, and hBarH fields set properly. }
355:
356: { LOCAL TO UNIT }
357:
358:
359:   VAR
360:
361:     lineHeight, margin, wHeight, wWidth : INTEGER;
362:     fInfo : FontInfo;
363:
364:   BEGIN
365:     IF GetTUHandle( wPtr ) THEN
366:       BEGIN
367:         GetFontInfo( fInfo );
368:         WITH theTUHandle^^, wPtr^.portRect DO
369:           BEGIN
370:             lineHeight := textHeight( wPtr );
371:             margin := ( blankMargin + blankMargin ) + ScBarWidth;
372:             wHeight := ( bottom - top ) - ( margin + fInfo.leading );
373:             wWidth := ( right - left ) - margin;
374:             linesPerPage := wHeight DIV lineHeight;
375:             SetRect( vRect, 0, 0, wWidth, linesPerPage * lineHeight );
376:             OffsetRect( vRect, blankMargin, blankMargin );
377:             dRect := vRect;
378:             IF textWidth > wWidth
379:               THEN dRect.right := textWidth
380:             END { with }
381:           END { if }
382:         END; { FormatText }
383:
384:
385: FUNCTION TUAttach;
386:
387: { Returns TRUE if a new TUREc can be allocated and attached to wPtr.
388:   If TRUE, procedure stores the TUHandle in the window's refCon field. }
389:
390:   VAR
391:
392:     error : BOOLEAN;
393:     teH : TEHandle;
394:     dRect, vRect : Rect;
395:
396:
397:   PROCEDURE AddControls;
398:
399:   { Add vertical and horizontal scroll bars to wPtr and to TUREc. }
400:
401:   VAR
402:
403:     cv, ch : ControlHandle;
404:
405:   BEGIN
406:     MakeNewScroll( wPtr, 'V', cv );
407:     MakeNewScroll( wPtr, 'H', ch );
408:     WITH theTUHandle^^ DO
409:       BEGIN
410:         vBarH := cv; hBarH := ch
411:       END { with }
412:     END; { AddControls }
413:
414:

```

(continued)


```

415: BEGIN { TUAttach }
416:
417:     error := TRUE;      { Unless all that follows succeeds }
418:
419:     theTUHand := TUHandle( NewHandle( SizeOf( TUREc ) ) );
420:     WindowPeek( wPtr )^.refCon := LONGINT( theTUHand );
421:     IF theTUHand <> NIL THEN
422:     BEGIN
423:         theTUHand^.textWidth := columnInches * oneInch;
424:         AddControls;          { Add scroll bars }
425:         FormatText( wPtr, dRect, vRect ); { Calc linesPerPage & rects }
426:         teH := TENew( dRect, vRect );    { Allocate TEREc }
427:         IF teH = NIL
428:         THEN
429:             BEGIN { Deallocate TUREc and exit with error }
430:                 DisposHandle( Handle( theTUHand ) );
431:                 WindowPeek( wPtr )^.refCon := LONGINT( NIL )
432:             END
433:         ELSE
434:             BEGIN { Initialize remaining fields and exit with no error }
435:                 WITH teH^^ DO
436:                     IF paragraphs
437:                     THEN crOnly := 0      { cr = end of paragraph }
438:                     ELSE crOnly := -1;    { cr = end of line }
439:                     theTUHand^.textRecH := teH; { Assign the TEREc handle }
440:                     error := FALSE
441:             END { else }
442:         END; { if / with }
443:
444:         TUAttach := NOT error { Return function result }
445:
446:     END; { TUAttach }
447:
448:
449: PROCEDURE TUDispose;
450:
451: { Disposes of the TUREc associated with this window. Call this procedure
452: before closing or disposing a window variable. You may reuse the window,
453: however, and even pass it again to TUAttach to display other text. }
454:
455: BEGIN
456:     IF GetTUHandle( wPtr ) THEN
457:     BEGIN
458:         TEDispose( theTUHand^.textRecH ); { Dispose TEREc record }
459:         KillControls( wPtr );              { Dispose all controls }
460:         DisposHandle( Handle( theTUHand ) ); { Dispose TUREc record }
461:         WindowPeek( wPtr )^.RefCon := LONGINT( NIL ); { For safety }
462:         theTUHand := NIL
463:     END { if }
464:     END; { TUDispose }
465:
466:
467: PROCEDURE TUClick;
468:
469: { Handles scrolling for mouse down events inside this window's scroll
470: bars. Call this procedure when you receive a MouseDown event and when
471: FindWindow indicates the location was in a control. Assumes wPtr
472: is the current port. }
473:
474: VAR
475:
476:     theControl : ControlHandle;
477:     partCode   : INTEGER;
478:

```

```

479: BEGIN
480:   IF GetTUHandle( wPtr ) THEN
481:     BEGIN
482:       GlobalToLocal( where );
483:       partCode :=
484:         FindControl( where, wPtr, theControl );
485:       IF ( theControl <> NIL ) THEN
486:         WITH theTUHand^^ DO
487:           IF ( theControl = vBarH ) OR { If control is a horiz }
488:             ( theControl = hBarH ) THEN { or vert bar, then... }
489:             IF partCode = inThumb THEN { If manually thumbing, }
490:               BEGIN { ...then scroll to new page }
491:                 partCode := TrackControl( theControl, where, NIL );
492:                 PageText
493:               END ELSE { ...else do scroll buttons }
494:                 partCode := TrackControl( theControl, where, @ScrollText )
495:             END { if }
496:           END; { TUClick }
497:
498:
499: PROCEDURE TUUpdate;
500:
501: { Updates text displayed in window. Call this procedure in response to an
502: update event. It's your responsibility to draw controls associated with
503: the window, however. This procedure updates only the text. }
504:
505: BEGIN
506:   IF GetTUHandle( wPtr ) THEN
507:     WITH theTUHand^^ DO
508:       TEUpdate( wPtr^.portRect, textRech )
509:   END; { TUUpdate }
510:
511:
512: PROCEDURE TUAActivate;
513:
514: { Activates (Activate=TRUE) or deactivates (Activate=FALSE) the controls
515: associated with this window. }
516:
517:
518: PROCEDURE DoHilite( cHand : ControlHandle; hiliteState : INTEGER );
519:
520: { Hilite or UnHilite cHand control unless NIL }
521:
522: BEGIN
523:   HiliteControl( cHand, hiliteState );
524:   ValidRect( cHand^.ctrlRect )
525: END; { DoHilite }
526:
527:
528: BEGIN
529:   IF GetTUHandle( wPtr ) THEN
530:     BEGIN
531:       IF Activate THEN
532:         BEGIN
533:           DoHilite( theTUHand^^.vBarH, 0 ); { Activate control }
534:           DoHilite( theTUHand^^.hBarH, 0 )
535:         END ELSE
536:         BEGIN
537:           DoHilite( theTUHand^^.vBarH, 255 ); { Deactivate control }
538:           DoHilite( theTUHand^^.hBarH, 255 )
539:         END { else }
540:       END { if }
541:     END; { TUAActivate }
542:
543:

```

(continued)


```

610:                BEGIN
611:                teLength := len;           { Save length in Terec }
612:
613:                TEdText( textRecH ); { WARNING: can compact heap! }
614:                error := FALSE           { Tell caller all is okay }
615:                END { with }
616:            END; { if }
617:            IF FSClose( fileNum ) <> noErr { Close text file }
618:            THEN error := TRUE;
619:            IF NOT error THEN
620:            BEGIN
621:                SetScBarValues( 0, 0 ); { Set text to extreme top left }
622:                InvalRect( wPtr^.portRect ) { Force display of text }
623:            END { if }
624:            END; { if }
625:            TureadText := NOT error      { Report function result }
626:            END; { TureadText }
627:
628:
629: END. { TextUnit }

```

Listing 5.8. READER.R

```

1: *-----*
2: * Reader.PAS resources -- Compile with RMaker *
3: *-----*
4:
5: Programs:Windows.F:Reader.RSRC ;; Send output to here
6:
7:
8: *-----*
9: * About box string list *
10: *-----*
11: TYPE STR# ;; String list resource
12: ,1 (32) ;; Resource ID and attribute (purgeable)
13: 6 ;; Number of strings that follow
14: Text Reader ;; Program name
15: by Tom Swan ;; Author
16: Version 1.00 ;; Version number
17: (C) 1987 by Swan Software ;; Copyright notice
18: P. O. Box 206, Lititz, PA 17543 ;; Address
19: (717)-627-1911 ;; Telephone
20:
21:
22: *-----*
23: * The Apple Info menu *
24: *-----*
25:
26: TYPE MENU
27: ,1
28: \14
29: About Reader...
30: (-
31:
32:
33: *-----*
34: * The File menu *
35: *-----*
36:

```

(continued)

```

37: TYPE MENU
38:   ,2
39: File
40:   Quit /Q
41:
42:
43: *-----*
44: * The Edit menu *
45: *-----*
46:
47: TYPE MENU
48:   ,3
49: Edit
50:   (Undo /Z
51:   (-
52:   (Cut /X
53:   (Copy /C
54:   (Paste /V
55:   (Clear
56:
57:
58: *-----*
59: * Window template *
60: *-----*
61:
62: TYPE WIND
63:   ,1 (32)           ;; ID number and attribute (purgeable)
64: Instructions        ;; Window title
65: 48 10 335 502       ;; top, left, bottom, right coordinates
66: Visible NoGoAway    ;; Visible window without close button
67: 8                   ;; Std document window with grow and zoom boxes
68: 0                   ;; Window reference (none)
69:
70:
71: * END

```

Listing 5.9. READER.PAS

```

1: {$O Programs:Windows.F: }           { Send compiled code to here }
2: {$R Programs:Windows.F:Reader.Rsrc} { Use this compiled resource file }
3: {$U-}                               { Turn off standard library units }
4:
5:
6: PROGRAM Reader;
7:
8: (*
9:
10:  * PURPOSE : Read a text file.  Demonstrate TextUnit tools.
11:  * SYSTEM  : Macintosh / Turbo Pascal
12:  * AUTHOR  : Tom Swan
13:
14: *)
15:
16:
17: {$U Programs:Units.F:MacExtras }      { Open these library unit files }
18: {$U Programs:Units.F:TextUnit }
19:
20:

```

```

21:  USES
22:
23:      Memtypes, QuickDraw, OSIntf, ToolIntf, PackIntf,
24:      MacExtras, TextUnit;
25:
26:
27:
28:  CONST
29:
30:      FileID      = 2;      { File menu Resource ID and commands }
31:      QuitCmd     = 1;
32:
33:      FileName = 'ReadMe';  { Must be on same volume as program }
34:
35:      WindowID    = 1;      { Program's window resource ID number }
36:
37:
38:  VAR
39:
40:      wRec        : WindowRecord;      { Program's window data record }
41:      wPtr        : WindowPtr;         { Pointer to above wRec }
42:      watch       : CursHandle;        { Handle to wrist watch cursor }
43:      quitRequested : BOOLEAN;         { TRUE if quitting }
44:
45:
46:
47:  PROCEDURE DoKeyPress( ch : CHAR );
48:
49:  { Do something with an incoming character }
50:
51:      BEGIN
52:      END; { DoKeyPress }
53:
54:
55:  PROCEDURE DoMouseClicked( whichWindow : WindowPtr );
56:
57:  { Process mouse clicks inside windows }
58:
59:      BEGIN
60:          TUClick( whichWindow, theEvent.where )
61:      END; { DoMouseClicked }
62:
63:
64:  PROCEDURE DoCloseRequest;
65:
66:  { Window go-away button was clicked or menu close command selected. }
67:
68:      BEGIN
69:          quitRequested := TRUE
70:      END; { DoCloseRequest }
71:
72:
73:  PROCEDURE DoFileMenuCommands( cmdNumber : INTEGER );
74:
75:  { Execute File menu command }
76:
77:      BEGIN
78:          IF cmdNumber = QuitCmd
79:              THEN quitRequested := TRUE
80:          END; { DoFileMenuCommands }
81:
82:
83:  PROCEDURE DoEditMenuCommands( cmdNumber : INTEGER );
84:
85:  { Execute Edit menu command }
86:

```

(continued)

```

87:   BEGIN
88:       IF NOT SystemEdit( cmdNumber - 1 ) THEN { ignore command }
89:       END; { DoEditMenuCommands }
90:
91:
92:   PROCEDURE DoCommand( command : LongInt );
93:
94:   { Execute a menu command }
95:
96:       VAR
97:
98:           whichMenu    : INTEGER;      { Menu number of selected command }
99:           whichItem     : INTEGER;      { Menu item number of command }
100:
101:   BEGIN
102:
103:       whichMenu := HiWord( command );    { Find the menu }
104:       whichItem := LoWord( command );    { Find the item }
105:
106:       CASE whichMenu OF
107:
108:           AppleID      : DoAppleMenuCommands( whichItem );
109:           FileID        : DoFileMenuCommands( whichItem );
110:           EditID        : DoEditMenuCommands( whichItem )
111:
112:       END; { case }
113:
114:       HiliteMenu( 0 ) { Unhighlight menu title }
115:
116:   END; { DoCommand }
117:
118:
119:   PROCEDURE DrawScrollBars( whichWindow : WindowPtr );
120:
121:   { Draw v & h scroll bars and other controls }
122:
123:       VAR
124:
125:           vBarRect      : Rect;        { Vertical scroll bar }
126:           hBarRect      : Rect;        { Horizontal scroll bar }
127:           gbRect        : Rect;        { Grow box }
128:
129:   BEGIN
130:       DrawGrowIcon( whichWindow );
131:       DrawControls( whichWindow );
132:       CalcControlRects( whichWindow, hBarRect, vBarRect, gbRect );
133:       ValidRect( hBarRect );
134:       ValidRect( vBarRect );
135:       ValidRect( gbRect )
136:   END; { DrawScrollBars }
137:
138:
139:   PROCEDURE DrawContents( whichWindow : WindowPtr );
140:
141:   { Display window contents }
142:
143:   BEGIN
144:       EraseRect( whichWindow^.portRect );
145:       DrawScrollBars( whichWindow );
146:       TUUpdate( whichWindow )
147:   END; { DrawContents }
148:
149:
150:   PROCEDURE MouseDownEvents;
151:

```

```

152: { Someone pressed the mouse button. Check its location and respond. }
153:
154:   VAR
155:
156:       partCode : INTEGER;      { Identifies what item was clicked. }
157:
158:   BEGIN
159:
160:       WITH theEvent DO
161:
162:           BEGIN
163:
164:               partCode := FindWindow( where, whichWindow );
165:
166:               CASE partCode OF
167:
168:                   inMenuBar
169:                       : DoCommand( MenuSelect( where ) );
170:
171:                   inSysWindow
172:                       : SystemClick( theEvent, whichWindow );
173:
174:                   inContent
175:                       : IF whichWindow <> FrontWindow
176:                           THEN SelectWindow( whichWindow )
177:                           ELSE DoMouseClicked( whichWindow );
178:
179:                   inDrag
180:                       : DragTheWindow( whichWindow, where );
181:
182:                   inGrow
183:                       : IF whichWindow <> FrontWindow
184:                           THEN
185:                               SelectWindow( whichWindow )
186:                           ELSE
187:                               BEGIN
188:                                   ResizeWindow( whichWindow, theEvent.where );
189:                                   TUREsize( whichWindow )
190:                               END; { else }
191:
192:                   inGoAway
193:                       : IF TrackGoAway( whichWindow, where )
194:                           THEN (* DoClose *);
195:
196:                   inZoomIn, InZoomOut
197:                       : IF TrackBox( whichWindow, where, partCode ) THEN
198:                           BEGIN
199:                               ZoomInOut( whichWindow, partCode );
200:                               TUREsize( whichWindow )
201:                           END { if }
202:
203:                   END { case }
204:
205:               END { with }
206:
207:           END; { MouseDownEvents }
208:
209:
210:   PROCEDURE KeyDownEvents;
211:
212:   { A key was pressed. Do something with incoming character. }
213:
214:   VAR
215:
216:       ch : CHAR;
217:

```

(continued)


```

218: BEGIN
219:     WITH theEvent DO
220:     BEGIN
221:
222:         ch := CHR( BitAnd( message, charCodeMask ) ); { Get character }
223:
224:         IF BitAnd( modifiers, CmdKey ) <> 0 { If command key pressed }
225:         THEN DoCommand( MenuKey( ch ) ) { then execute command }
226:         ELSE DoKeyPress( ch ) { else use character }
227:
228:     END { with }
229: END; { KeyDownEvents }
230:
231:
232: PROCEDURE UpdateEvents;
233:
234: { Part or all of a window requires redrawing }
235:
236: VAR
237:
238:     oldPort : GrafPtr; { For saving / restoring port }
239:
240: BEGIN
241:     GetPort( oldPort ); { Save current port }
242:     whichWindow :=
243:         WindowPtr( theEvent.message ); { Extract window pointer }
244:     SetPort( whichWindow ); { Change current grafPort }
245:     BeginUpdate( whichWindow ); { Calculate new visRgn }
246:     DrawContents( whichWindow ); { Draw/redraw window contents }
247:     EndUpdate( whichWindow ); { Reset original visRgn }
248:     SetPort( oldPort ); { Restore old port }
249: END; { UpdateEvents }
250:
251:
252: PROCEDURE ActivateEvents;
253:
254: { Activate or deactivate windows }
255:
256: BEGIN
257:     WITH theEvent DO
258:     BEGIN
259:
260:         whichWindow := WindowPtr( message ); { Extract window pointer }
261:         SetPort( whichWindow ); { Change current port }
262:
263:         DrawScrollBars( whichWindow ); { Draw bars & grow box }
264:
265:         IF BitAnd( modifiers, activeFlag ) <> 0 THEN
266:         BEGIN
267:             FixEditMenu( FALSE ); { Activate a window }
268:             TUActivate( whichWindow, TRUE )
269:         END ELSE
270:         BEGIN
271:             FixEditMenu( TRUE ); { Deactivate a window }
272:             TUActivate( whichWindow, FALSE )
273:         END { else }
274:
275:     END { with }
276: END; { ActivateEvents }
277:
278:
279: PROCEDURE SetUpMenuBar;
280:
281: { Initialize and display menu bar }
282:

```

```

283: BEGIN
284:
285:     appleMenu := GetMenu( AppleID ); { Read menu resources }
286:     fileMenu  := GetMenu( FileID  );
287:     editMenu  := GetMenu( EditID  );
288:
289:     InsertMenu( appleMenu, 0 ); { Insert into menu list }
290:     InsertMenu( fileMenu,  0 );
291:     InsertMenu( editMenu,  0 );
292:
293:     AddResMenu( appleMenu, 'DRVr' ); { Add desk accessory names }
294:
295:     DrawMenuBar { Display the menu bar }
296:
297: END; { SetUpMenuBar }
298:
299:
300: PROCEDURE SetUpWindow;
301:
302: { Initialize this program's window record }
303:
304: CONST
305:
306:     paragraphs = TRUE; { Text formatted into paragraphs in window }
307:     columnInches = 6; { Width of text in inches }
308:
309: BEGIN
310:     wPtr := GetNewWindow( WindowID, @Wrec, POINTER( -1 ) );
311:     SetPort( wPtr );
312:     TextFont( Geneva );
313:     IF NOT TUAttach( wPtr, paragraphs, columnInches )
314:     THEN ExitToShell
315: END; { SetUpWindow }
316:
317:
318: PROCEDURE Initialize;
319:
320: { Program calls this routine one time at start }
321:
322: BEGIN
323:
324:     SetUpMenuBar; { Initialize and display menus }
325:     DisplayAboutBox; { Identify program }
326:     SetUpWindow; { Initialize window variables }
327:
328:     watch := GetCursor( WatchCursor );
329:     quitRequested := FALSE;
330:
331:     SetCursor( watch^^ );
332:     IF NOT TureadText( wPtr, fileName, 0 )
333:     THEN ExitToShell;
334:     InitCursor
335:
336: END; { Initialize }
337:
338:
339: FUNCTION QuitConfirmed : BOOLEAN;
340:
341: { The program's "deinitialization" routine. }
342: { Returns TRUE if it's okay to quit program }
343:
344: BEGIN
345:     QuitConfirmed := quitRequested
346: END; { QuitConfirmed }
347:
348:

```

(continued)

```

349: PROCEDURE DoSystemTasks;
350:
351: { Do operations at each pass through main program loop }
352:
353:   BEGIN
354:
355:       SystemTask;      { Give DAs their fair share of time }
356:
357:       IF FrontWindow = NIL THEN
358:
359:           BEGIN { Set up menu commands for empty desktop }
360:
361:               FixEditMenu( FALSE )
362:
363:           END ELSE
364:
365:           IF FrontWindow <> wPtr THEN
366:
367:               BEGIN { Set up menu commands for active desk accessory }
368:
369:                   FixEditMenu( TRUE )
370:
371:               END { else / if }
372:
373:           END; { DoSystemTasks }
374:
375: BEGIN
376:
377:   Initialize;
378:
379:   REPEAT
380:
381:       DoSystemTasks;
382:
383:
384:       IF GetNextEvent( everyEvent, theEvent ) THEN
385:
386:           CASE theEvent.what OF
387:
388:               MouseDown    : MouseDownEvents;
389:               KeyDown      : KeyDownEvents;
390:               AutoKey       : { ignored };
391:               UpdateEvt     : UpdateEvents;
392:               ActivateEvt   : ActivateEvents
393:
394:           END { case }
395:
396:   UNTIL QuitConfirmed
397:
398: END.

```

Reader Play-by-Play

READER.PAS (Listing 5.9, 1-43)

TextUnit greatly simplifies the tedious job of displaying text in windows and handling scroll bars. If you compare Listing 5.9 with ApShell, you'll see only a few changes. TextUnit takes care of most details but, to make it work properly, you need to follow several rules.

The first step is to include both units **MacExtras** and **TextUnit** as in lines 17–24. You also must add the five standard units in your program's **USES** declaration as the listing shows. Similar to other simple examples that use only one window, **Reader** declares its window variables permanently in the application's global memory space (40–41). For multiple windows, follow the plan outlined in Listing 5.2 earlier in this chapter.

DoKeypress to DoCommand (47–116)

Procedure **DoKeypress** (47–52) does nothing here. In your own program, you can add whatever you need to do in response to typing. **DoMouseClicked** (55–61) adds a call to **TextUnit** tool **TUClick**, passing the appropriate window pointer and the **where** field of the event record. **TUClick** processes mouse clicks in scroll bars, paging text up and down as well as left and right. It fully implements auto scrolling, letting you hold the mouse down in the scroll bar arrows to move text continuously in one direction or another. The rest of the procedures in this section are similar to those in **ApShell**.

DrawScrollBars to MouseDownEvents (119–207)

DrawScrollBars is unchanged from **ApShell**. **DrawContents** adds a single call to **TextUnit** tool **TUUpdate** (146), passing the window pointer as a parameter. **TUUpdate** redraws text inside the window borders in response to update events.

MouseDownEvents (150–207) adds two calls to tool **TUResize** when the window size changes. When you click and drag the window's grow box in the lower right corner, the procedure first calls **ResizeWindow** to change the window's borders (188), and then calls **TUResize** to adjust text and scroll borders inside the new boundaries (189). Similarly, lines 196–201 respond to clicks in the window's zoom box, if it has one. After zooming the window in or out (199), **TUResize** adjusts text and scroll bars to match.

KeyDownEvents to END (210–398)

KeyDownEvents and **UpdateEvents** (210–249) are unchanged. **ActivateEvents** adds calls to tool **TUActivate** (268,272), passing the window pointer and **TRUE** if the window is becoming active or **FALSE** if inactive. This dims or highlights the window's scroll bars.

SetUpMenuBar (279–297) is unchanged. **SetUpWindow** (300–315) shows how to prepare a window for displaying text. First, it loads the resource template and displays the window (310). It then sets the current port and text font (311–312). You can change the text font, size, and style to anything you want—**TextUnit** works with all text variations. Line 313 is most important. **TUAttach** prepares the window to receive text, specifying whether to reformat paragraphs and to what size. Set constant **paragraphs** to **FALSE** to display program text where each line ends in a car-

riage return. Set it **TRUE** to display paragraphs ending in returns. For program listings, change the text font to Monaco, the same as the Turbo editor uses.

Adjust constant **columnInches** (307) to the size you want TextUnit to limit text lines. This size can be smaller or larger than the window width, although when **paragraphs** is **TRUE**, the display looks best if the size is about the same. If larger, you can scroll text horizontally; otherwise, TextUnit deactivates horizontal scrolling. Notice that the size is in inches, assuming that one pixel is 1/72-inch square. For full size windows and text in paragraphs, six inches is the correct value to use. For program listings with no paragraphs, change this value to 14 or larger.

Procedure **Initialize** (318–336) shows how to read a text file and attach it to the prepared window. It also demonstrates how to change the cursor shape to a wrist watch, telling people to be patient while the program reads from disk. Variable **watch** is of type **CursHandle** (see line 42), a handle to a relocatable memory block containing the bit pattern for this shape. The program loads the shape into memory by calling **GetCursor** (328), passing constant **WatchCursor** as a parameter.

Reading text files into windows is simple with TextUnit. First, the program changes the cursor to a watch (331) and then calls **TUReadText**, passing the prepared window pointer from **SetUpWindow**, a file name, and a volume reference number. If the function returns **FALSE**, then it was not able to read the text file and the program ends (333). In this example, the volume number 0 specifies the same volume from which you ran the program. In Chapter 6, you'll learn how to reference files in other volumes. Line 334 ends the procedure by resetting the cursor back to its standard arrow shape.

As you can see, adding text to windows requires little effort on your part. TextUnit handles nearly every detail involving scroll bars, text formatting, and display. The next section describes its tools in detail although you don't need to study every line in order to use them.

TextUnit Play-by-Play

Refer to Listing 5.7 for the following notes about using TextUnit tools. The unit declares three data types (29–39), following the methods for creating handles described at the beginning of this chapter. When you attach text to windows, TextUnit stores a handle to the text inside the window record. This method avoids having to maintain extra variables in your program.

TUHandle (29) is a pointer to master pointer type **TUPtr**, which addresses **TURec**, the data object that TextUnit stores on the heap in a relocatable memory block. Inside that record are five fields (34–38). The first, **textRecH**, is another handle, which the toolbox **TextEdit** routines define. This handle locates the actual characters as stored in memory, also in relocatable blocks.

Two more handles, **hBarH** and **vBarH**, locate the scroll bar controls as stored on the heap. These too are also in relocatable memory blocks. Finally, two miscellaneous integers, **linesPerPage** and **textWidth**, record facts about the text that the unit needs for some of its operations.

You can see that text in memory is a complex, intertwined set of memory blocks, all of which the program locates by using handles to relocatable areas on the heap. To keep track of which structure goes with which window, TextUnit stores the handle to **TURec** in the window's **refCon** field, a 32-bit **LONGINT** variable the toolbox reserves for your use. By doing this, the window record itself keeps track of the handle to its **TURec**, which in turn keeps track of the handles to the text in memory and the two scroll bars associated with this window. Figure 5.9 illustrates how this structure appears in memory. You might like to know that I drew this diagram *before* writing the program. When dealing with multiple handles and structures in memory, it helps to have a good mental image of the relationship between every part. A good diagram helps develop that image.

Examining the diagram, you may notice a few elements that TextUnit does not describe. For example, **TERec** is TextEdit's own text record object, which has yet another handle locating the actual text in memory. Look at the window record.

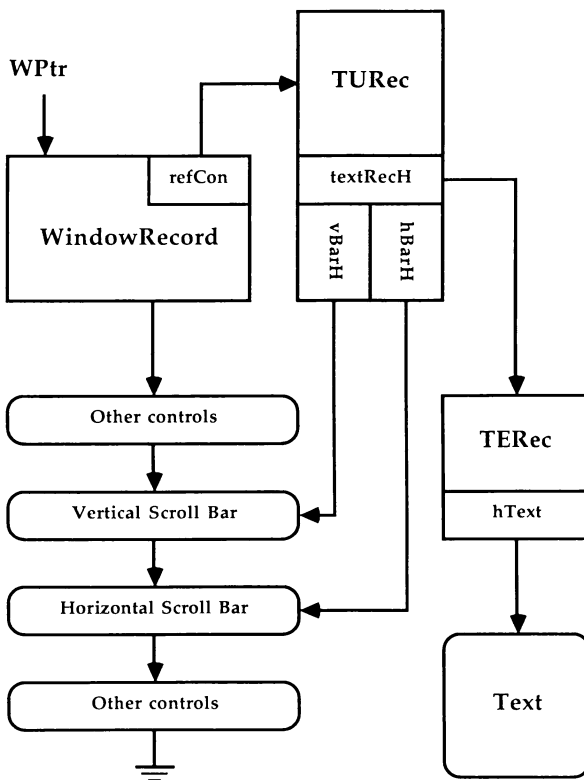


Figure 5.9 TextUnit (Listing 5.7) organizes objects in memory, according to this diagram. When designing complex, interrelated structures, a drawing like this one is priceless.

Its **refCon** field stores the handle to TextUnit's **TURec**, but it also points to a list of controls that include the same ones TextUnit finds via **hBarH** and **vBarH**.

You don't need to worry about every element in all of these structures. In fact, the diagram here has more detail than it needs to describe how TextUnit works. But you should be aware that this and other structures have many relocatable parts and pieces in memory, all linked in one way or another on the heap. As you can see, it's important that you avoid fragmenting the heap and that you carefully program your own handles according to the rules at the beginning of this chapter. Remember that the structures you invent compete for memory with other objects.

TextUnit declares four local constants you can change to affect how the unit operates. These constants (68–71) are visible only to routines in the unit. You cannot use them in your programs. Constant **overLap** specifies the number of lines that repeat when you page text up and down. If you have 12 lines in a window, and **overLap** is 4, then paging up and down moves 8 lines at a time. Overlapping lines this way gives people a reference when paging. If you want no overlapping, set **overLap** to zero.

Constant **blankMargin** controls the amount of white space between the window borders and characters. Usually the default value of 4 makes a good looking display, but you can change this value if you want. Set **colWidth** to the number of pixels you want to scroll horizontally every time you click in the horizontal scroll bar's arrows. Small values scroll more slowly; larger values more quickly. Don't change **oneInch**. It specifies the number of pixels in an inch and corresponds with the Macintosh's 1/72-inch square pixel dimensions. If a future model should change that specification, you can adjust TextUnit to match or make **oneInch** variable to handle different configurations.

Variable **theTUHand** (76) holds the handle to the currently active **TURec**. TextUnit sets this handle to a copy of the window's **refCon** field to gain a little speed and avoid repeated references to the window record.

The following describes routines internal to TextUnit. You cannot call these routines in your own programs, but you might want to know how they operate. After that, the chapter ends with a description of each of the TextUnit tools that are available for using in programs.

GetTUHandle to CheckNoScroll (80–204)

Function **GetTUHandle** (80–90) extracts from a window record the handle to a **TURec** record. Line 88 may appear confusing at first. It peeks at the window record to get to its **refCon** field, where TextUnit previously stored the **TURec** handle. It then casts that value, a **LONGINT** data type, into a **TUHandle** and assigns it to **theTUHand**. Finally, the function sets its value to **TRUE** or **FALSE** depending on whether the handle is **NIL**. If it is, then this window was not properly prepared, and the handle must not be used.

The next procedure (93–140) resizes scroll bars attached to a window. Set **barType** to 'H' for horizontal scroll bars or to 'V' for verticals. The procedure operates

by first hiding the scroll bar control (115), making it temporarily invisible. It then calculates the bar's new dimensions (118–133), moves it to a new location (135), and changes its size (136). Because this also draws the scroll bar in the window, line 139 validates its enclosing rectangle, avoiding flutter during updates.

MakeNewScroll (143–170) calls **ResizeScrollBar** (166) after creating a new control. **NewControl** (156–162) creates the control as a relocatable object on the heap. The three zeros at line 160 set the minimum, maximum, and current values that work together to position the thumb box inside the bar. When you later attach text to the window, **TextUnit** calculates the actual values according to how much text is in memory.

CheckNoScroll (173–204) helps **TextUnit** decide whether to allow scrolling. For example, run the program and shrink the window. You should see the horizontal scroll bar become active. Click the zoom box or expand the window to full size. The scroll bar again becomes inactive because the full width of text is now visible in the window. Notice how line 186 double dereferences two handles in a **WITH** statement. This is perfectly acceptable because the following assignments cannot cause the Memory Manager to relocate memory blocks on the heap.

Lines 196–202 check the current control value. If zero, meaning the thumb box is at the top or far left, the procedure decides whether to disable the scroll bar by setting its maximum value to zero. In the case of horizontal scrolls, this happens if the text viewing width (how much you see left and right) is less than the text destination width (the maximum width of a line). For vertical scrolls, it happens if the total number of text lines is less than the lines per one page (**lpp**).

SetScBarValues to PageText (207–281)

SetScBarValues takes two parameters, **horizValue** and **vertValue**. These values position the thumb box inside the scroll bar to a position relative to the amount of text in memory and the lines now on display. The toolbox Control Manager draws the thumb box and calculates where to place it. Our procedure simply sets the control's minimum values to zero (227,236); the vertical bar maximum to the number of lines (228); and the horizontal bar maximum to the line width (237). While this is happening, it also hides the controls (226,235) and then reshows them (230,239) to avoid screen flutter when the thumb box moves.

One important feature of **SetScBarValues** is the way it locks the relocatable block containing **TURec**, which in turn contains the handles to the two scroll bars. **HLock** (221) temporarily prevents the Memory Manager from moving the block associated with **theTUHand**. It must do this because the **WITH** statement at line 223 double dereferences the handle to get to the **vBarH** and **hBarH** fields in that record and because several of the procedures called inside the **WITH** statement might cause the Memory Manager to shuffle relocatable blocks. This is the only place in **TextUnit** that locks the heap, by the way. If you never want it to do so, rewrite **SetScBarValues** to use one of the methods described earlier for working with relocatable memory blocks.

SetScBarValues's final job is to call **CheckNoScroll** (245), testing whether the new scroll bar values require disabling the control. It is at this time that **TextUnit** deactivates a scroll bar when you expand a window, displaying the full text width or height.

PageText (250–281) displays text in a window by scrolling to a specific location equal to the value of both the horizontal and vertical scroll bars. In this case, there's no need to lock the heap even though the **WITH** statement double dereferences two handles (263). Procedure **GetCtlValue** cannot cause memory-shuffling and, therefore, the program may call it without worrying about the Memory Manager moving a relocatable block. But **TEScroll** (275) might cause a reshuffling. Doesn't this break the rule? The answer is both yes and no. The previous **WITH** statement is invalid after the call to **TEScroll**—if you refer to any field at **theTUHand**^^ or **textRecH**^^ after line 275, you risk damaging the heap. But prior to calling **TEScroll**, there is no danger.

PageText calls **CheckNoScroll** to check whether to disable a control (279). To see why it is necessary to do this here, run the program and shrink the window to about one quarter screen size. Scroll text horizontally until the thumb box is approximately in the center of its travel. Now expand the window to full screen and scroll to bring the thumb box far left. The scroll bar should disable at this point. Take out line 279 and repeat the experiment to see the difference. After expanding the window and scrolling, the bar remains active even though the full text width is visible.

ScrollText (284–347)

TextUnit never directly calls **ScrollText**. Instead, it passes the address of this procedure to the Control Manager, which calls it to enable automatic scrolling while you hold down the mouse inside one of the scroll bar arrows or in its gray region. (See line 494.) Although it seems complicated, the procedure is not hard to understand.

First, it copies the current settings of the control that's being scrolled (304–306). It then calculates two values, **lineColFull** and **pageFull**, setting these to the number of lines to scroll up or down for vertical scroll bars or the number of pixels to move left or right for horizontal bars. Notice how line 309 tests whether the control handle passed to this procedure equals **vBarH**, the field in the **TURec** at **theTUHand**^^. When you need to find out whether a structure equals one object or another, compare their handles as shown here. If two handles are equal, they refer to the same object in memory.

The **CASE** statement (321–339) then sets variable **amount** equal to the total amount of scrolling required for the **partCode** passed to the procedure. The **partCode** indicates in which part of the scroll bar the mouse arrow points. If the thumb box is against the end of a bar, the **partCode** does not equal one of the four values in the **CASE** statement (**inUpButton** to **inPageDown**), and **amount** will be zero, avoiding an annoying flutter that happens in some programs when you hold the mouse button down in a scroll bar after the thumb box bumps into one end.

The actual scrolling takes place in the **IF** statement (341–345). The procedure sets the control value to its current value plus **amount** and calls **PageText** to display text at this new position.

FormatText (350–382)

The final local procedure is **FormatText**, which calculates various fields associated with a **TURec** attached to a window. The assignments are obvious if you take the time to read them carefully. Notice how it sets **dRect** and **vRect**, the two rectangles that control the way text appears in the window. Variable **dRect** is the *destination rectangle*. It determines the widest line length. The toolbox Text-Edit routines consider the destination rectangle to be bottomless—its width is more important than its height. Variable **vRect**, or *view rectangle*, determines the viewable portion of the window in which text appears. Here, we set **vRect**'s **right** field to the maximum text width or to the window's width, whichever is greater.

The destination rectangle specifies the text margins, determining the widest line. The view rectangle specifies how much text you can see at one time. If the destination and view rectangles are equal, then you see all the text there is (left to right). If the view rectangle is smaller than the destination, then to see an entire line requires scrolling horizontally. If the reverse is true—the view rectangle is larger than the destination—then text will have large white spaces in its left and right margins.

This ends the local procedures in TextUnit. The next section describes the tools that you can call directly in your programs. For reference, the procedure declarations and parameter lists are repeated.

```
FUNCTION TUAttach( wPtr : WindowPtr;
  paragraphs : BOOLEAN; columnInches :
  INTEGER ) : BOOLEAN;
```

Call **TUAttach** (385–446) after creating a new window record, passing its pointer in parameter **wPtr**. Set **paragraphs TRUE** if you want TextUnit to consider carriage returns to mark the ends of paragraphs. Set it **FALSE** if you want carriage returns to mark line ends, as they do in program listings. Integer parameter **columnInches** equals the width of the widest line and can be larger or smaller than the window width. If larger, TextUnit enables horizontal scrolling.

The procedure begins by creating a relocatable memory block on the heap to hold a **TURec** record (419–420). It then stores the block's handle in the window record's **refCon** field, converting it to a **LONGINT** type in the process. (Handles and long integers are both 32-bits long and are therefore compatible.) To protect against errors, the procedure tests **theTUHand** (421). If **NIL**, then **NewHandle** was not able to create the memory block and the function returns **FALSE**.

As long as the handle was not **NIL**, lines 423–441 complete the initialization. First, the procedure calculates the text width, calls a sub procedure to add scroll bars, and formats the text as explained earlier (423–425). The sub procedure, **Add-**

Controls, calls **MakeNewScroll** (406–407) twice and assigns the resulting handles **ch** and **cv** to the scroll bar fields in the **TextUnit** record.

With scroll bars attached, line 426 allocates space for text in memory, calling **TextEdit** function **TENew** with the destination and view rectangles that **Format-Text** calculated earlier. If this does not work, lines 430–431 immediately dispose the **TextUnit** handle and set the window's **refCon** field to **NIL**. Dealing with errors in multiple-handle structures such as this can be tricky. As shown here, be sure you don't inadvertently leave partial objects on the heap. (This is a time when diagrams like Figure 5.9 are worth their weight in pixels.)

A **WITH** statement (435–438) sets field **crOnly** according to the value of **paragraphs**, telling the toolbox whether to format lines into paragraphs (0) or leave them as they are in program listings (–1). This field is in the **TextEdit**'s text record, which contains various other items that don't concern us here. (See *Inside Macintosh* for the complete definition under type **TERec**.) Finally, the function saves the handle to the text record in the **TURec**'s **textRecH** field (439) and sets the error flag to **FALSE**.

```
PROCEDURE TUDispose( wPtr : WindowPtr );
```

TUDispose (449–464) reverses what **TUAttach** does. Call it when you're done using a window with an attached **TextUnit** record. Do this whether you created the window record on the heap or as a local variable in your program as in Listing 5.9 (40). After disposing, you can use the window for any purpose. You can also call **TUAttach** again to prepare to attach more text, perhaps from a different file.

To completely erase all structures associated with the window, the procedure calls **TEDispose** (458), disposing the **TERec** that stores various **TextEdit** parameters and the actual text in memory (see Figure 5.9). Next, **KillControls** disposes all controls associated with the window, including the two scroll bars but also any other controls you attached via other means. Be aware of this side effect of calling **TUDispose**. After that, **DisposeHandle** erases the **TURec** from the heap, leaving only the window record behind (460). For safety, lines 461–462 set **refCon** and **theTUHand** to **NIL**, guarding against accidentally using those handles in the future.

The steps in disposing a multiple-handle structure are simpler than the steps to create one—but they are equally critical. Again, a diagram such as the one in Figure 5.9 is invaluable for knowing exactly what to dispose and in what order. It would be a mistake, for example, to dispose of **TERec** *after* **TURec**, a fact the diagram makes clear.

```
PROCEDURE TUClick( wPtr : WindowPtr;
  where : Point );
```

Call **TUClick** (467–496) to process mouse down events in windows. Pass the window pointer in parameter **wPtr** and the **where** field from the event record. See Listing 5.9 (60) for an example. The procedure converts the global coordinate of

the mouse pointer to a value local to the window (482) and then calls **FindControl** to locate in exactly which part of the window the mouse pointer points.

As long as this position is inside of a control, lines 486–494 call **TrackControl** in one of two ways. If you are manually moving the scroll bar's thumb box, line 491 uses **NIL** as the last parameter. This tells **TrackControl** to move the thumb box to a new position, ending only when you release the mouse button. In this case, **PageText** (492) then displays the text from this new position.

But if you hold the mouse down in either an arrow or in the gray region to one side of the thumb box, line 494 calls **TrackControl** with the address of procedure **ScrollText**, described earlier. **TrackControl** itself calls **ScrollText** to activate automatic scrolling, which continues to the end of the text or until you release the mouse. You might want to re-examine **ScrollText** and **PageText** at this point to be certain you understand how **TextUnit** scrolls text in windows.

```
PROCEDURE TUUpdate( wPtr : WindowPtr );
```

TUUpdate (499–509) is simple. It calls the toolbox **TextEdit** routine **TEUpdate**, passing the window's **portRect** field and the handle to the text record, **textRecH**. Call **TUUpdate** as part of your update event handler. (See Listing 5.9, line 146 for example.)

```
PROCEDURE TUAActivate( wPtr : WindowPtr;
    activate : BOOLEAN );
```

TUAActivate (512–541) highlights or dims scroll bars depending on whether window **wPtr** is becoming active (**activate** is **TRUE**) or not. Sub-procedure **DoHilite** (518–525) does the actual highlighting and dimming. It calls **HiliteControl**, passing the control handle and state—either 0 to activate or 255 to deactivate the control. The **ValidRect** that follows avoids flutter for update and activate event pairs as in other procedures that directly draw items in windows.

```
PROCEDURE TUResize( wPtr : WindowPtr );
```

This procedure (544–582) reforms the scroll bars after the window size changes. Call **TUResize** any time you shrink or expand a window.

The procedure demonstrates another technique that avoids locking the heap. The **WITH** statement (565–570) copies three handles out of **TURec**, saving their values in the three local variables, **cv**, **ch**, and **teH**. It then uses those variables in calls to procedures that might cause the Memory Manager to move the relocatable block associated with the record. Copying the handles means the program always finds the objects to which they refer—even though the original handle variables (**vBarH**, for example) might move around with **TURec**.

Line 579 shows the correct way to force a redisplay of the window's contents. Invalidating the view rectangle causes an update event, which eventually calls **TUUpdate** to draw the text in its new size.

```
FUNCTION TureadText( wPtr : WindowPtr;
  fileName : Str255; volNum : INTEGER ) : BOOLEAN;
```

The final TextUnit routine, **TureadText** (585–629), does what its name implies. It reads a text file into memory, associating that text with the window record that you previously prepared by calling **TUAttach**. If the function returns **TRUE**, then reading was successful—otherwise an error occurred, usually because **fileName** does not exist or because there is not enough memory to hold its text. Set parameter **volNum** to zero to read the file from the current directory. Or, pass the volume reference number of another directory to read files from there. (The next chapter shows how to use volume numbers.)

The function cannot use standard Pascal I/O operations such as **Reset** and **ReadLn** to read text. Instead, it reads characters as a block, inserting them as quickly as possible into memory. This requires additional work, but the extra speed is worth the effort.

Line 600 opens the file by calling **FSOpen**. If that function returns **noErr**, then line 602 moves to the end of the file, setting parameter **len** to the number of characters the file contains. After that, **SetFPos** resets the file to its beginning to prepare to read it from disk. Use these two functions as shown here whenever you need to determine a file's size.

Line 605 copies the text handle, the one that locates the area reserved in memory for storing characters, to avoid locking the heap during what follows. **SetHandleSize** (606) then tells the Memory Manager to expand that relocatable memory block to a size large enough to hold **len** characters. If this doesn't work, the manager reserves as much memory as it can. Because of this, line 607 assigns back to **len** the actual size of the memory block at handle **hCopy**. This potentially chops the ends of very large files but still lets you view at least most of their text. Because TextUnit only reads and displays text files, this does no harm.

The actual file reading takes place at line 608 with a call to **FSRead**. If that function returns **noErr**, the following **WITH** statement saves the new text length in the **TERec** field **teLength** (611) and calls **TECalText**, passing the text record's handle. **TECalText** calculates an internal array of pointers to the beginning of each line to make scrolling and paging run fast. Because this action might rearrange the heap, the preceding **WITH** statement dereferencing the two handles is invalid after line 613.

Final steps close the file, return **TRUE** or **FALSE** as the function value, and set scroll bar values initially to zero (621). Then, **InvalRect** forces an update event of the window's **portRect**, displaying the text just loaded into memory.

To save space here, examples do not follow the full ApShell design in Chapter 4. Despite that, there are a few unavoidable duplications. You can save a little typ-

ing time by starting new examples with copies of previous ones. Except for the final example, programs do not let you use desk accessories, do not have cut and paste Edit menus, and do not display About Program boxes. Instead, a single File menu usually contains one or two commands that demonstrate specific dialog features. Of course, you can use the same techniques in complete Macintosh programs such as those in Chapters 4, 5, and 7.

STANDARD FILE DIALOGS

Any program that uses disk files should let people choose file names in the standard way that has become a Macintosh trademark. Figures 6.1 and 6.2 show the two standard file (SF) dialog windows that every Macintosh owner knows by heart.

Figure 6.1 is the dialog you normally see after choosing a program's Open command. Use it to prompt for names of files to open and read. The second SF dialog in Figure 6.2 is similar but includes at bottom left an entry area for typing file names. Use this dialog in response to your program's Save as and Save commands.

An example demonstrates how to program the standard file dialogs in the two figures. Before typing it in, you must type in and compile to disk Listing 6.18, `DIALOGUNIT.PAS`, which contains various dialog tools that all examples in this chapter use. The listing is near the end of this chapter on page 324. Next, type in Listing 6.1, save as `SF.R`, and compile with `RMaker` to create the program's resource file. Then type in Listing 6.2, save as `SF.PAS`, and compile with `Turbo`.

When you run the program, you'll see two File menu commands, Open and

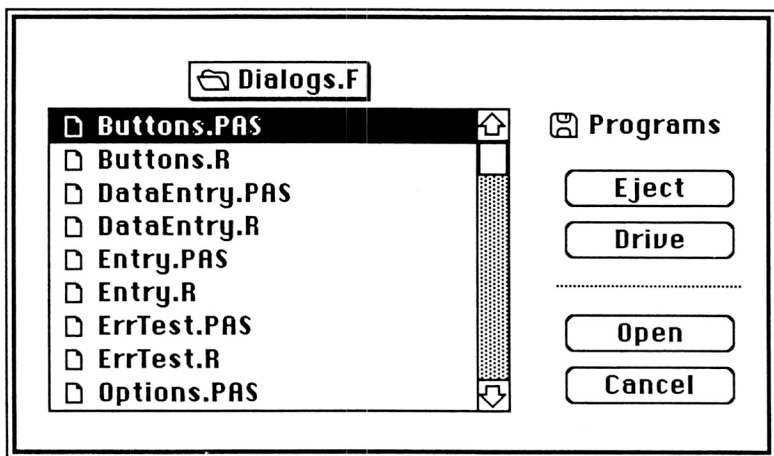


Figure 6.1 Use the standard file dialog to prompt for existing file names.

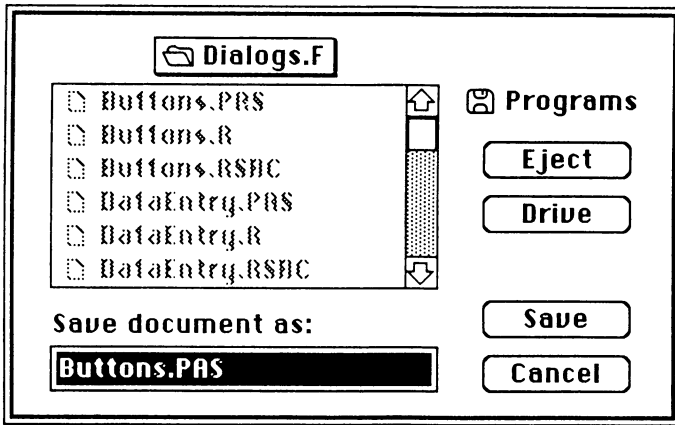


Figure 6.2 Use this alternate standard file dialog to prompt for new file names, displaying a warning (not shown here) if a file of that name already exists.

Save, which demonstrate the standard dialog windows. Feel free to experiment—you cannot change or harm any files on disk, even when the program requests whether you want to replace an existing file. The SF dialogs never open or create files themselves—they merely prompt you for file names that programs then use for these operations.

Listing 6.1. SF.R

```

1: *-----*
2: * SF.PAS resources -- Compile with RMaker      *
3: *-----*
4:
5: Programs:Dialogs.F:SF.RSRC      ;; Send output to here
6:
7:
8: *-----*
9: * The File menu                    *
10: *-----*
11:
12: TYPE MENU
13:   ,1                                ;; Menu ID number to use in program
14: File                               ;; Menu title as shown in menu bar
15:   Open
16:   SaveAs
17:   Quit
18:
19:
20: * END

```


Listing 6.2. SF.PAS

```

1: {$O Programs:Dialogs.F: }           { Send compiled code to here }
2: {$R Programs:Dialogs.F:SF.Rsrc}     { Use this compiled resource file }
3: {$U-}                               { Turn off standard library units }
4:
5:
6: PROGRAM SF;
7:
8: (*
9:
10:  * PURPOSE : Standard File Dialogs
11:  * SYSTEM  : Macintosh / Turbo Pascal
12:  * AUTHOR   : Tom Swan
13:
14:  *)
15:
16:
17: {$U Programs:Units.F:DialogUnit }    { Open this library unit file }
18:
19:
20:     USES
21:
22:         Memtypes, QuickDraw, OSIntf, ToolIntf, PackIntf, DialogUnit;
23:
24:
25:
26:     CONST
27:
28:         FileID      = 1;      { File menu Resource ID and commands }
29:         OpenCmd      = 1;
30:         SaveAsCmd    = 2;
31:         QuitCmd      = 3;
32:
33:
34:     VAR
35:
36:         fileMenu      : MenuHandle;    { Handle to program's only menu }
37:         fileName      : Str255;        { Current file name }
38:         theEvent       : EventRecord;   { Events from operating system }
39:         whichWindow    : WindowPtr;     { Window applying to event }
40:         quitRequested  : BOOLEAN;       { TRUE if quitting }
41:
42:
43:
44: PROCEDURE DoOpen;
45:
46: { Demonstrate how to use standard file dialog to prompt for
47:   "open" file names }
48:
49:     VAR
50:
51:         reply : SFReply;    { File information }
52:
53:     BEGIN
54:         IF GetFileName( reply, 'TEXT' ) THEN
55:             BEGIN
56:                 SysBeep( 3 );
57:                 fileName := reply.fName
58:             END { if }
59:
60:     END; { DoOpen }
61:

```

```

62: PROCEDURE DoSaveAs;
63:
64: { Demonstrate how to use standard file dialog to prompt for
65:   "save as" file names }
66:
67:   CONST
68:
69:       savePrompt = 'Save document as: ';
70:
71:   VAR
72:
73:       reply : SFReply;
74:
75:   BEGIN
76:       IF MakeFileName( reply, savePrompt, fileName ) THEN
77:           BEGIN
78:               SysBeep( 3 );
79:               fileName := reply.fName
80:           END { if }
81:       END; { DoSaveAs }
82:
83:
84: PROCEDURE DoCommand( command : LongInt );
85:
86: { Execute a menu command }
87:
88:   VAR
89:
90:       whichMenu   : INTEGER;    { Menu number of selected command }
91:       whichItem   : INTEGER;    { Menu item number of command }
92:
93:   BEGIN
94:
95:       whichMenu := HiWord( command );    { Find the menu }
96:       whichItem := LoWord( command );    { Find the item }
97:
98:       IF whichMenu = FileID THEN
99:           CASE whichItem OF
100:               OpenCmd      : DoOpen;
101:               SaveAsCmd    : DoSaveAs;
102:               QuitCmd      : quitRequested := TRUE
103:           END; { case }
104:       HiliteMenu( 0 ) { Unhighlight menu title }
105:
106:   END; { DoCommand }
107:
108:
109: PROCEDURE MouseDownEvents;
110:
111: { Someone pressed the mouse button.  Check its location and respond. }
112:
113:   VAR
114:
115:       partCode : INTEGER;    { Identifies what item was clicked. }
116:
117:   BEGIN
118:       WITH theEvent DO
119:           BEGIN
120:               partCode := FindWindow( where, whichWindow );
121:               CASE partCode OF
122:                   inMenuBar
123:                       : DoCommand( MenuSelect( where ) )
124:               END { case }
125:           END { with }
126:       END; { MouseDownEvents }
127:
128:

```

(continued)

```

129: PROCEDURE Initialize;
130:
131: { Program calls this routine one time at start }
132:
133:   BEGIN
134:
135:     InitGraf( @thePort );
136:     InitFonts;
137:     InitWindows;
138:     InitMenus;
139:     TEInit;
140:     InitDialogs( NIL );
141:     InitCursor;
142:     FlushEvents( everyEvent, 0 );
143:
144:     fileMenu := GetMenu( FileID );
145:     InsertMenu( fileMenu, 0 );
146:     DrawMenuBar;
147:
148:     quitRequested := FALSE;
149:
150:     fileName := ''
151:
152:   END; { Initialize }
153:
154:
155: BEGIN
156:   Initialize;
157:   REPEAT
158:     SystemTask;
159:     IF GetNextEvent( everyEvent, theEvent ) THEN
160:       CASE theEvent.what OF
161:        MouseDown      : MouseDownEvents;
162:       END { case }
163:   UNTIL quitRequested
164: END.

```

SF Play-by-Play

Refer to Listing 6.2 for the notes that follow. Line 37 declares a global string variable, **fileName**, which holds whatever name you select with either SF dialog. Procedure **Initialize** sets **fileName** to a null string (150) when the program starts.

Procedures **DoOpen** (44–59) and **DoSaveAs** (62–81) call procedures in DialogUnit (Listing 6.18) to display the two SF dialogs. Both procedures declare a variable, **reply**, of type **SFReply** (see Figure 6.3), that fully describes a file on disk. Although this example saves only the file name field **fName**, a program could save the entire **SFReply** record for all the files it uses.

Field **good** is **TRUE** if the rest of the fields in **SFReply** are valid. If this field is **FALSE**, then someone clicked the Cancel button and the program should take an appropriate action—usually ignoring the Open or Save command. Field **copy** has no purpose—it’s a leftover from earlier designs. Even so, don’t attempt to use this or any other unused fields in toolbox records. Apple Company programmers might reinstate such fields in the future. The same is true of fields marked “reserved.” They’re not reserved for you or me!

TYPE

```

SFReply =
RECORD
    good      : BOOLEAN;      { TRUE if remaining fields are valid }
    copy      : BOOLEAN;      { unused }
    fType     : OSType;        { New file names only }
    vRefNum   : INTEGER;       { Volume reference number }
    version   : INTEGER;       { Version number (always 0) }
    fName     : String[63]     { File name }
END; { SFReply }

```

Figure 6.3 The toolbox **SFReply** record fully describes a file on disk.

Similarly, field **version** is always zero and has no useful purpose. Never set it to another value or expect the toolbox to keep track of file versions for you through this field.

Integer **vRefNum** is the volume reference number, referring to the disk and folder that contains the file. Some file commands—**Reset** and **Rewrite**, for example—take only a file name. Others like **FSOpen** in the toolbox File Manager take both a file name and volume reference. When using file-name-only routines, you might have to set the default volume to **vRefNum** before referencing a file. To do this, use a statement like the following. It sets the default volume to **vRefNum** to prepare for **Reset**, which looks only on the current volume for the file name you pass as the second parameter.

```

IF SetVol( NIL, reply.vRefNum ) = NoErr
THEN Reset( f, reply.fName );

```

Returning to Listing 6.2, procedure **DoOpen** (44–59) passes an uninitialized **reply** record to **GetFileName** along with a four-character string indicating the file type (54). The example specifies ‘TEXT’, limiting file names in the SF dialog window to text files. Table 6.1 lists other file types for several popular programs. Try replacing ‘TEXT’ with some of these names and rerun the example program to see how to limit the dialog window to specific file types.

Table 6.1 File types for several popular programs.

Program	File type
1st Base	1STD
FactFinder	FACT
Filevision	PICB
MacDraw	DRWG
MacPaint	PNTG
MacTerminal	TEXT
MacWrite	WORD
MegaFiler	MFIL
Multiplan	TEXT,MPBN
Think Tank	TEXT
Turbo Pascal	TEXT

If **GetFileName** (54) returns **TRUE**, **DoOpen** beeps (56) and sets global **fileName** to the reply's **fName** field. (The beep just lets you know the test works.)

DoSaveAs (62–81) works similarly. **MakeFileName** (76) returns **TRUE** if the fields in **reply** indicate that someone clicked the Save button (see Figure 6.2) or pressed the Return key to select a name for a new file. If a file of the same name already exists, the toolbox File Manager takes care of requesting permission to overwrite it—you do not have to do that in your program. If **MakeFileName** is **TRUE**, then **vRefNum** and **fName** fields in **reply** are filled in, ready for use.

MakeFileName (76) takes three parameters, the **SFReply** record **reply**, a string, and a file name. The string is the prompt you want to see above the edit box in Figure 6.2. The file name is the current name, darkened as in the figure. If there is no current name, pass a null string (' ') as the third parameter.

Both **MakeFileName** and **GetFileName** simplify using the two SF dialogs. The play-by-play for **DialogUnit** (Listing 6.18) explains other ways to program these dialogs but, for most programs, the method in the example is the easiest.

DIALOG ITEM LISTS

You create custom dialog designs as resources that programs load into memory. A dialog resource is a template, a pattern for the toolbox Dialog Manager to create various structures in memory that it uses to display and manipulate the dialog. There are two parts to such a template: the dialog definition (DLOG) and the dialog item list (DITL). The DLOG locates the dialog window and tells the Window Manager how big to make it. The DITL contains a list of items such as button controls and text associated with the dialog.

For an example of creating a simple dialog window with an item list, type in Listing 6.3, save as **BUTTONS.R**, and compile with **RMaker**. Also type in Listing

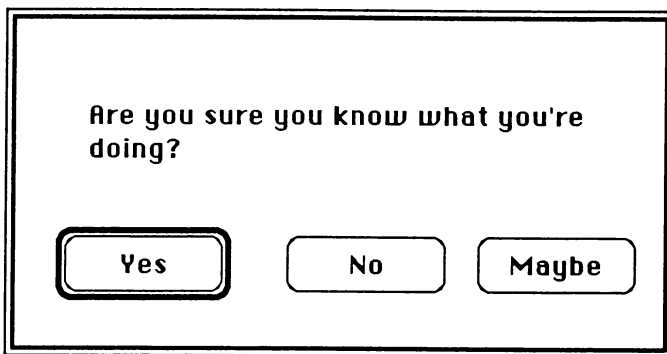


Figure 6.4 The dialog window of Listing 6.4. Buttons.

6.4 and save as BUTTONS.PAS. Figure 6.4 shows the dialog window you see when you choose the File menu's Buttons command.

Listing 6.3. BUTTONS.R

```

1: *-----*
2: * Buttons.PAS resources -- Compile with RMaker      *
3: *-----*
4:
5: Programs:Dialogs.F:Buttons.RSRC ;; Send output to here
6:
7:
8: *-----*
9: * The File menu                                     *
10: *-----*
11:
12: TYPE MENU
13:     ,1                      ;; Menu ID number to use in program
14: File                        ;; Menu title as shown in menu bar
15:     Buttons
16:     Quit
17:
18:
19: *-----*
20: * The buttons dialog                                 *
21: *-----*
22:
23: TYPE DLOG
24:     ,1000                   ;; Resource ID number
25: Buttons                     ;; Title (not displayed)
26: 100 100 250 400             ;; Top, Left, Bottom, Right
27: Visible NoGoAway           ;; Immediately visible, no go-away box
28: 1                           ;; Std double-border dialog window
29: 0                           ;; Reference value (none)
30: 1000                        ;; ID of dialog item list (following)
31:
32:
33: *-----*
34: * The buttons dialog item list                       *
35: *-----*
36:
37: TYPE DITL
38:     ,1000 (32)              ;; Resource ID, (32)=purgeable
39: 4                            ;; Number of items following
40:
41: BtnItem Enabled              ;; 1. Yes button
42: 100 20 127 95
43: Yes
44:
45: BtnItem Enabled              ;; 2. No button
46: 100 125 127 200
47: No
48:
49: BtnItem Enabled              ;; 3. Maybe button
50: 100 215 127 290
51: Maybe
52:
53: StatText Disabled            ;; 4. Text
54: 35 30 76 270
55: Are you sure you know what you're doing?
56:
57: * END

```

Listing 6.4. BUTTONS.PAS

```

1: {$O Programs:Dialogs.F: }           { Send compiled code to here }
2: {$R Programs:Dialogs.F:Buttons.Rsrc} { Use this compiled resource file }
3: {$U-}                               { Turn off standard library units }
4:
5:
6: PROGRAM Buttons;
7:
8: (*
9:
10:  * PURPOSE : Dialog buttons demo
11:  * SYSTEM   : Macintosh / Turbo Pascal
12:  * AUTHOR   : Tom Swan
13:
14:  *)
15:
16:
17: {$U Programs:Units.F:DialogUnit }    { Open this library unit file }
18:
19:
20:     USES
21:
22:         Memtypes, QuickDraw, OSIntf, ToolIntf, PackIntf, DialogUnit;
23:
24:
25:
26:     CONST
27:
28:         FileID      = 1;      { File menu Resource ID and commands }
29:         ButtonsCmd   = 1;
30:         QuitCmd      = 2;
31:
32:         DialogID     = 1000;  { Resource ID of buttons dialog }
33:
34:
35:     VAR
36:
37:         fileMenu      : MenuHandle;    { Handle to program's only menu }
38:         theEvent       : EventRecord;   { Events from operating system }
39:         whichWindow    : WindowPtr;     { Window applying to event }
40:         quitRequested  : BOOLEAN;       { TRUE if quitting }
41:
42:
43:
44: PROCEDURE DoButtons;
45:
46: { Demonstrate using multiple buttons in a dialog }
47:
48:     CONST
49:
50:         Yes   = 1;      { Item list numbers for these buttons }
51:         No    = 2;
52:         Maybe = 3;
53:
54:     VAR
55:
56:         dp : DialogPtr;
57:         itemHit : INTEGER;
58:

```

```

59: BEGIN
60:   dp := GetNewDialog( DialogID, NIL, POINTER(-1) );
61:   IF dp <> NIL THEN
62:     BEGIN
63:       OutlineOk( dp );
64:       REPEAT
65:         ModalDialog( NIL, itemHit )
66:       UNTIL itemHit IN [ Yes, No, Maybe ];
67:       IF itemHit = Yes
68:       THEN SysBeep( 3 );
69:       DisposDialog( dp )
70:     END { if }
71:   END; { DoButtons }
72:
73:
74: PROCEDURE DoCommand( command : LongInt );
75:
76: { Execute a menu command }
77:
78:   VAR
79:
80:     whichMenu      : INTEGER;      { Menu number of selected command }
81:     whichItem      : INTEGER;      { Menu item number of command }
82:
83:   BEGIN
84:
85:     whichMenu := HiWord( command ); { Find the menu }
86:     whichItem := LoWord( command ); { Find the item }
87:
88:     IF whichMenu = FileID THEN
89:       CASE whichItem OF
90:         ButtonsCmd      : DoButtons;
91:         QuitCmd         : quitRequested := TRUE
92:       END; { case }
93:       HiliteMenu( 0 ) { Unhighlight menu title }
94:
95:     END; { DoCommand }
96:
97:
98:
99: PROCEDURE MouseDownEvents;
100:
101: { Someone pressed the mouse button. Check its location and respond. }
102:
103:   VAR
104:
105:     partCode : INTEGER; { Identifies what item was clicked. }
106:
107:   BEGIN
108:     WITH theEvent DO
109:       BEGIN
110:         partCode := FindWindow( where, whichWindow );
111:         CASE partCode OF
112:           inMenuBar
113:             : DoCommand( MenuSelect( where ) )
114:         END { case }
115:       END { with }
116:     END; { MouseDownEvents }
117:
118:
119: PROCEDURE Initialize;
120:
121: { Program calls this routine one time at start }
122:

```

(continued)


```

123:   BEGIN
124:
125:       InitGraf( @thePort );
126:       InitFonts;
127:       InitWindows;
128:       InitMenus;
129:       TEInit;
130:       InitDialogs( NIL );
131:       InitCursor;
132:       FlushEvents( everyEvent, 0 );
133:
134:       fileMenu := GetMenu( FileID );
135:       InsertMenu( fileMenu, 0 );
136:       DrawMenuBar;
137:
138:       quitRequested := FALSE
139:
140:   END; { Initialize }
141:
142:
143: BEGIN
144:   Initialize;
145:   REPEAT
146:       SystemTask;
147:       IF GetNextEvent( everyEvent, theEvent ) THEN
148:           CASE theEvent.what OF
149:               MouseDown      : MouseDownEvents;
150:           END { case }
151:       UNTIL quitRequested
152:   END.

```

Buttons Play-by-Play

BUTTONS.R (1-58)

The resource text file (Listing 6.3) shows how to design a standard dialog window with an item list containing three buttons and a line of text. Lines 23–30 declare a resource of type DLOG and assign it the ID number 1000, which the program then uses to load this resource into memory. You can use any positive integer value as the ID, but values above 128 avoid conflicts with dialogs that belong to other processes. (Most such IDs are negative so you could probably use 1, 2, 3, and so on. Using high values like 1000 guarantees against any possible conflicts.)

Line 25 gives the dialog a title but, unlike window titles, this text never appears on screen. If you're pressed for space, you could type a single character like X or D. The four integers at line 26 list the top, left, bottom, and right coordinate values, specifying both the size and location of the dialog window. These values are global, meaning they refer to the Macintosh screen usually with (0,0) in the upper left corner.

Line 28 selects the window style for the dialog's border. The value 1 specifies the double-border window that all the examples in this chapter use. You can use a different style if you want (the plain box 2 or alternate box 3 are good choices). But tempting as it is to stylize programs with unusual dialog windows, remember that many people, especially those who become nervous around computers, often feel more comfortable with programs that use standard designs.

The value in line 29 has no meaning. Similar to the window record's **refCon** field (see Chapter 5), you can store whatever you want in this 32-bit field. Line 30 contains the ID of the dialog's item list, declared elsewhere in the resource file. It's usually best to use the same ID number for the dialog (24), its reference to the item list (30), and for the item list itself (38). But you can use different IDs or even the same value for two different dialogs that you want to have the same items—buttons, text, and so on. That way, two or more dialogs share the same items but set various controls, buttons, and boxes according to one situation or another.

The item list for this example declares three buttons and a line of text. Its definition begins with **TYPE DITL** (37) followed by the same resource ID used in the **DLOG** definition—here at line 30. The value 32 in parentheses makes the item list purgeable, meaning the Memory Manager is free to remove or purge the items from memory to make room for other purposes. Always make dialog item lists purgeable. When the Dialog Manager loads the item list resource, it creates a *copy* of it in memory. If you do not specify the original resource to be purgeable, you cause two copies of every item to be stored in memory although you need only one.

The third line of the dialog item list (39) tells how many items follow. Here there are three buttons (**BtnItem**), each enabled. An enabled item is active. Programs can detect when someone clicks the mouse pointer inside an enabled item. They cannot detect clicks in disabled ones. Each item has four coordinate values—in top, left, bottom, right order—that are relative (local) to the dialog window. After designing a dialog, to shift it left six pixels requires modifying only the dialog coordinates (line 26 in the example). Because the item locations are relative to the dialog's own coordinates, they automatically follow the dialog window no matter where you position it. The third line in each button (43,47,51) is the text you want to display inside the button outlines. Match these labels with the buttons in Figure 6.4.

The fourth item is static text (**StatText**), disabled at line 53. If you enable the text, then someone can click the mouse on it as though it were a button or a check box. Most of the time, declare **StatText** items Disabled unless you want this to happen.

BUTTONS.PAS (1–152)

Procedure **DoButtons**, (44–71) in Listing 6.4, shows how to display a dialog along with its item list. It begins with three integer constants, **Yes**, **No**, and **Maybe** (48–52) equal to the position of those buttons in the dialog's item list. To display the dialog, the program calls function **GetNewDialog** (60), passing the resource ID and two other parameters. The **NIL** tells **GetNewDialog** to place internal information belonging to this dialog on the heap. The **POINTER(–1)** parameter tells it to display the window in front of any others now on screen.

You can create your own dialog record as a local variable and pass its address to **GetNewDialog** in place of **NIL**. This avoids fragmenting the heap. To do this, add this variable between lines 57 and 58:

```
dRec : DialogRecord;
```

Then, change lines 60 and 69 as follows:

```
60: dp := GetNewDialog( DialogID, @dRec, POINTER(-1) );
69: CloseDialog( dp )
```

Because you create the dialog record as a local variable, pass its address to **GetNewDialog** as the parameter **@dRec**. When done using the dialog, close it by calling **CloseDialog**. Either method works, but passing **NIL** and letting the Dialog Manager create the dialog record itself is easier and saves stack space. In the previous chapter, you learned how doing this with regular windows can fragment the heap. But with modal dialogs like this one when nothing much else can happen while the dialog is visible, a temporarily fragmented heap is unlikely to cause any problems.

Line 63 calls the DialogUnit's **OutlineOK** procedure, passing the pointer (**dp**) returned by **GetNewDialog**. This draws a bold outline around the first object in the dialog's item list, usually the button with the label you're most likely to choose, typically Yes or Ok. The bold outline (see Figure 6.4) tells you that pressing the Return key as well as clicking with the mouse selects this button.

The **REPEAT** loop (64–66) calls **ModalDialog** with two parameters. The **NIL** tells the procedure to handle events in the usual way, returning value 1 in **itemHit** if you press the Return (or Enter) keys. The loop ends when **itemHit** is one of the three values, **Yes**, **No**, or **Maybe**, indicating a click in one of those buttons. Lines 67–68 sound a beep only if you click the **Yes** button (or press Return).

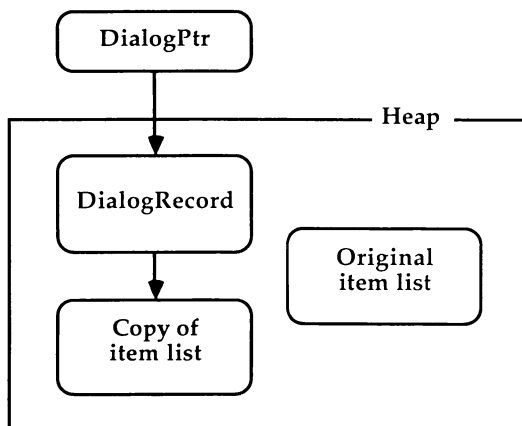


Figure 6.5 When you load a dialog, the toolbox copies its item list, leaving the original list floating in memory. For that reason, always make item list resources (type DITL) purgeable so the Memory Manager can remove them if it needs the space.

DIALOGS IN MEMORY

A dialog record contains a full window record (which contains a **GrafPort**) and is a most complex beast. For most uses, you can safely ignore its many fields and use them as this chapter describes. The complete record definition is in the *Guide* and *Inside Macintosh*.

More important than such details is the way the toolbox stores dialog records in memory when you call **GetNewDialog** as in the previous example. Figure 6.5 illustrates the relationship between your program's DialogPtr variable (**dp** in Listing 6.4) and items in memory.

As the figure shows, the **DialogPtr** variable points to a dialog record that **GetNewDialog** creates on the heap. This record in turn points to a copy of the dialog's item list. The original copy of this list remains in memory (the floating rounded box on the right). Because you mark it purgeable, though, if the memory manager needs the room, it automatically removes the original list. If you don't mark item lists purgeable, they permanently waste memory space.

In memory, item lists contain various structures, text, and controls such as buttons and scroll bars. Calling **DisposeDialog** releases the memory these items occupy along with the memory that the dialog record uses. But **DisposeDialog** does not release the memory that the original item list uses, making it doubly important not to forget to mark DITL resources purgeable.

ALERTS

Alerts are dialogs that usually contain only text and buttons. Use an alert to do just that—alert people to some condition or warn of the consequences of actions like quitting a program without saving a file.

Figure 6.6 shows one of the most common alert boxes, the result of quitting the program in Listing 6.6. The program assumes you made a change to a fictitious

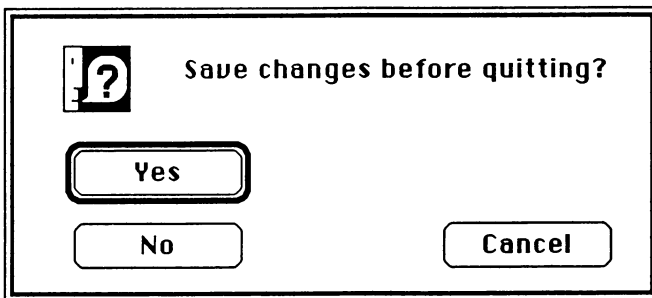


Figure 6.6 The dialog window of Listing 6.6, Save.



CautionAlert



NoteAlert



StopAlert

Figure 6.7 Alerts use one of these icons to call your attention to errors and notes.

file and, when you choose the Quit command, displays the alert asking you to respond in one of three ways: click Cancel to continue using the program, click No to throw away changes, or click Yes to save changes before quitting.

Alerts display the cartoon figure in the upper left corner of Figure 6.6. Instead of a question mark in the figure's speech balloon, you can display an exclamation or an asterisk (see Figure 6.7).

The next program shows how to add alerts to programs. Type in Listing 6.5, save as QUIT.R, and compile with RMaker. Then use Turbo to type in, compile, and run QUIT.PAS in Listing 6.6 to test the alert box in Figure 6.6.

Listing 6.5. QUIT.R

```

1: *-----*
2: * Quit.PAS resources -- Compile with RMaker *
3: *-----*
4:
5: Programs:Dialogs.F:Quit.RSRC      ;; Send output to here
6:
7:
8: *-----*
9: * The File menu *
10: *-----*
11:
12: TYPE MENU
13:   ,1
14: File
15:   Quit
16:
17:

```

```

18: *-----*
19: * The Save changes? alert *
20: *-----*
21:
22: TYPE ALRT
23:     ,1000 (4)           ;; Resource ID, (4) = preload
24:     73 105 198 400      ;; Top, Left, Bottom, Right
25:     1000                ;; Item list ID (following)
26:     5555                ;; Alert stages (none)
27:
28:
29: TYPE DITL               ;; Item list for alert (preceding)
30:     ,1000 (32)          ;; Resource ID, (32) = purgeable
31:     4                   ;; Number of items following
32:
33: BtnItem Enabled        ;; 1. Yes button
34:     60 25 82 105
35:     Yes
36:
37: BtnItem Enabled        ;; 2. Cancel button
38:     95 200 117 280
39:     Cancel
40:
41: BtnItem Enabled        ;; 3. No button
42:     95 25 117 105
43:     No
44:
45: StatText Disabled      ;; 4. Text with replaceable parameter (^0)
46:     14 76 34 285
47:     Save changes before ^0?
48:
49:
50: * END

```

Listing 6.6. QUIT.PAS

```

1: {$O Programs:Dialogs.F: }      { Send compiled code to here }
2: {$R Programs:Dialogs.F:Quit.Rsrc} { Use this compiled resource file }
3: {$U-}                          { Turn off standard library units }
4:
5:
6: PROGRAM Quit;
7:
8: (*
9:
10:  * PURPOSE : Quit with Save Changes? dialog
11:  * SYSTEM  : Macintosh / Turbo Pascal
12:  * AUTHOR   : Tom Swan
13:
14:  *)
15:
16:  USES
17:
18:      Memtypes, QuickDraw, OSIntf, ToolIntf, PackIntf;
19:
20:
21:
22:
23:  CONST
24:
25:      FileID      = 1;      { File menu Resource ID and commands }
26:      QuitCmd     = 1;

```

(continued)

```

27:
28:     SaveID          = 1000;  { Resource ID of Save changes? alert }
29:
30:     NullStr         = '';    { No blanks between the two quotes '' }
31:
32:
33:     VAR
34:
35:         fileMenu      : MenuHandle;      { Handle to program's only menu }
36:         theEvent       : EventRecord;     { Events from operating system }
37:         whichWindow    : WindowPtr;      { Window applying to event }
38:         quitRequested  : BOOLEAN;        { TRUE if quitting }
39:
40:
41:
42: PROCEDURE SaveChanges;
43:
44: { Dummy procedure }
45:
46:     BEGIN
47:         SysBeep( 3 );    { Two beeps = "changes saved" }
48:         SysBeep( 3 )
49:     END; { SaveChanges }
50:
51:
52: PROCEDURE DoCommand( command : LongInt );
53:
54: { Execute a menu command }
55:
56:     VAR
57:
58:         whichMenu      : INTEGER;        { Menu number of selected command }
59:         whichItem       : INTEGER;       { Menu item number of command }
60:
61:     BEGIN
62:
63:         whichMenu := HiWord( command );  { Find the menu }
64:         whichItem := LoWord( command );   { Find the item }
65:
66:         IF whichMenu = FileID THEN
67:             CASE whichItem OF
68:                 QuitCmd      : quitRequested := TRUE
69:             END; { case }
70:             HiliteMenu( 0 ) { Unhighlight menu title }
71:
72:         END; { DoCommand }
73:
74:
75: PROCEDURE MouseDownEvents;
76:
77: { Someone pressed the mouse button. Check its location and respond. }
78:
79:     VAR
80:
81:         partCode : INTEGER;    { Identifies what item was clicked. }
82:
83:     BEGIN
84:         WITH theEvent DO
85:             BEGIN
86:                 partCode := FindWindow( where, whichWindow );
87:                 CASE partCode OF
88:                     inMenuBar
89:                 : DoCommand( MenuSelect( where ) )
90:             END { case }
91:         END { with }
92:     END; { MouseDownEvents }

```

```

93:
94:
95: PROCEDURE Initialize;
96:
97: { Program calls this routine one time at start }
98:
99:   BEGIN
100:
101:     InitGraf( @thePort );
102:     InitFonts;
103:     InitWindows;
104:     InitMenus;
105:     TEInit;
106:     InitDialogs( NIL );
107:     InitCursor;
108:     FlushEvents( everyEvent, 0 );
109:
110:     fileMenu := GetMenu( FileID );
111:     InsertMenu( fileMenu, 0 );
112:     DrawMenuBar;
113:
114:     quitRequested := FALSE
115:
116:   END; { Initialize }
117:
118:
119: FUNCTION QuitConfirmed : BOOLEAN;
120:
121: { True if quitting and Yes button in Save? dialog alert clicked. }
122:
123:   CONST
124:
125:     Yes    = Ok;      { Yes button is first }
126:
127:   VAR
128:
129:     itemHit : INTEGER;
130:
131:   BEGIN
132:     IF quitRequested THEN
133:       BEGIN
134:         ParamText( 'quitting', nullStr, nullStr, nullStr );
135:         itemHit := CautionAlert( SaveID, NIL );
136:         IF itemHit = Yes
137:           THEN SaveChanges
138:           ELSE IF itemHit = Cancel
139:             THEN quitRequested := FALSE      { Cancel quit command }
140:         END; { if }
141:         QuitConfirmed := quitRequested
142:       END; { QuitConfirmed }
143:
144:
145:   BEGIN
146:     Initialize;
147:     REPEAT
148:       SystemTask;
149:       IF GetNextEvent( everyEvent, theEvent ) THEN
150:         CASE theEvent.what OF
151:          MouseDown    : MouseDownEvents;
152:         END { case }
153:     UNTIL QuitConfirmed
154:   END.

```


Quit Play-by-Play

QUIT.R (1-50)

The resource definition for an alert resembles a dialog. Line 22 starts a new resource of type **ALRT** and gives it an ID of 1000. As with dialog resource IDs, you can use any positive integer, but values greater than 128 guarantee against conflicts with resources the system uses for its own purposes.

The 4 in parentheses after the resource ID (23) marks this resource for preloading at the time the program runs. You don't have to preload alerts this way but it's usually a good idea, especially to display disk error messages. You might not be able to display the message, "Disk Directory is Unreadable," if, before showing it, you have to read it from the resource file on a damaged disk!

Line 24 lists the top, left, bottom, and right coordinate values describing the alert window's size and location. Unlike dialogs, though, you do not have a choice of window styles. Alerts always display as double-border windows (see Figure 6.6).

The 1000 in line 25 is the ID of the item list associated with this alert. An alert's item list is identical to a dialog's. In fact, dialogs and alerts could share the same lists. The 5555 at line 26 specifies this alert's *stages*, which you can use to change the alert based on the number of times an error or condition occurs. (This chapter does not use this feature, which some people find more confusing than helpful. The stages value of 5555 effectively turns the feature off. Consult *Inside Macintosh* for more details.)

The alert item list (type **DITL**) at lines 29-47 is nearly identical to the list in the previous example (see Listing 6.3, 37-55). Its resource ID (30) matches that in the alert definition and is marked purgeable. Of the four items that follow, three are enabled buttons (33-43) and one is static text (45-47).

Line 47 shows how to specify replaceable areas in strings. The $\wedge 0$ at the end of the message, "Save changes before $\wedge 0$?" lets the program pass a replacement string to appear at this location in the alert. With a simple change, therefore, Figure 6.6 can display the message, "Save changes before closing?" or "Save changes before transfer?" Any dialog or alert can have up to four such replaceable strings, $\wedge 0$, $\wedge 1$, $\wedge 2$, and $\wedge 3$. The next section explains how to use them.

QUIT.PAS (1-154)

Because this example doesn't actually write any data to disk, procedure **SaveChanges** (42-49) beeps twice to let you know the program calls it at the right time. When you quit the program, function **QuitConfirmed** (119-142) displays the alert box and lets you click the Yes, No, or Cancel buttons.

Line 134 shows how to substitute text in static text items. **ParamText** takes four parameters, the first corresponding to replaceable item $\wedge 0$, the second to $\wedge 1$, and so on. The example passes 'quitting' as the replacement for $\wedge 0$. Try other strings here and rerun the program. Set unused parameters to zero-length null strings as in the final three parameters at line 134.

ParamText replaces strings in the next alert or dialog the program displays. Therefore, always call **ParamText** to substitute strings for replaceable items *before* displaying a dialog or alert—even if you display it many times during the program. Remember that other procedures use the same technique to replace strings in their own alerts. If you don't reinitialize replacement strings before each use, you might display leftovers from other routines.

Line 135 displays the alert, rings the bell, and lets you click the Yes, No, or Cancel buttons. The function **CautionAlert** passes back the number of the clicked button. Clicking Yes or pressing Return sets **itemHit** to 1. Clicking No sets it to 2; Cancel to 3—the same numbers as the positions of these items in the alert's resource item list. Pass the alert resource ID as **CautionAlert**'s first parameter (**SaveID** in the example). The second parameter, **NIL**, tells the function to return 1 when you press Return or Enter, simulating the effect of clicking the first button. In this and other alerts, you can replace **NIL** with a pointer to a *filter*, a custom routine to replace the Dialog Manager's own programming for responding to keypresses and other events. Writing filters is an advanced subject not detailed here—consult *Inside Macintosh* for more information.

To see the other alert figures, replace **CautionAlert** in line 135 with **StopAlert** or **NoteAlert**. The only difference among the three is the symbol in the figure's speech balloon (Figure 6.7).

When using alerts, you do not have to create pointers, dialog records, or worry about fragmenting the heap. As this example shows, all you need is the resource and its ID. After using an alert, you do not have to close it or dispose the memory it occupies. The toolbox handles such details for you.

RADIO BUTTONS

In a group of radio buttons, punching one makes the others pop out like the buttons on a car radio or an old tape recorder. In dialogs, radio buttons make it easy to select one of several conditions—just point and click the button you want.

On screen, a radio button is a small circle, usually with a label to indicate what the button does. The punched-in button has a black dot inside; the others are white. Clicking one button turns another off. To make these actions easier to control, **DialogUnit** (Listing 6.18) contains routines and data types to organize radio buttons into groups.

Listing 6.8 demonstrates how to use these features. The program displays the dialog box in Figure 6.8 when you choose the Patterns command from the File menu. Clicking one of the five radio buttons to the left of the large rectangle changes its pattern. Clicking the Ok button removes the dialog window. Choosing Patterns again proves that the program remembers the previous setting—the pattern you most recently selected.

Type in Listing 6.7, save as **RADIO.R**, and compile with **RMaker**. Type in Listing 6.8 and save as **RADIO.PAS**. Then use **Turbo** to compile and run the test.

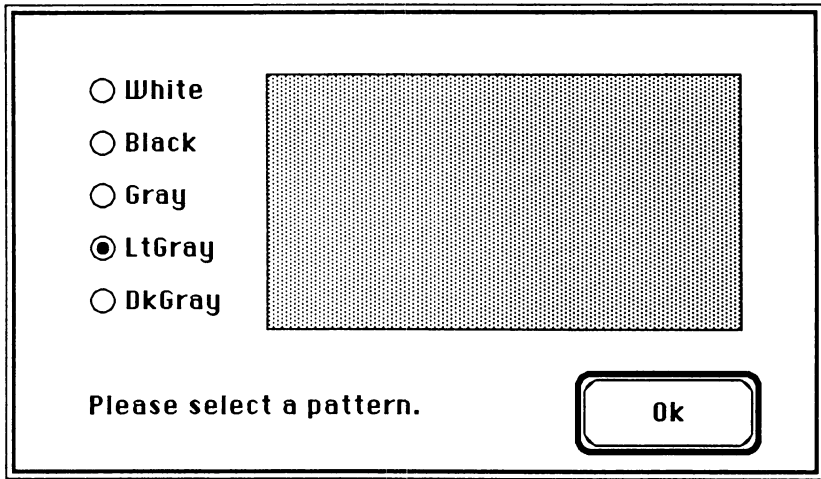


Figure 6.8 The dialog window of Listing 6.8, Radio.

Listing 6.7. RADIO.R

```

1:  *-----*
2:  * Radio.PAS resources -- Compile with RMaker      *
3:  *-----*
4:
5:  Programs:Dialogs.F:Radio.RSRC  ;; Send output to here
6:
7:
8:  *-----*
9:  * The File menu                                *
10: *-----*
11:
12: TYPE MENU
13:   ,1                      ;; Menu ID number to use in program
14: File                      ;; Menu title as shown in menu bar
15:   Patterns
16:   Quit
17:
18:
19: *-----*
20: * The radio buttons dialog                      *
21: *-----*
22:
23: type DLOG
24:   ,1000                  ;; Resource ID
25: Patterns                ;; Title (not displayed)
26: 76 71 286 442          ;; Top, Left, Bottom, Right
27: Visible NoGoAway       ;; Immediately visible, no go-away box
28: 1                      ;; Std double-border dialog window
29: 0                      ;; Reference value (none)
30: 1000                   ;; ID of dialog item list (following)
31:
32:

```

```

33: *-----*
34: * The radio buttons dialog item list *
35: *-----*
36:
37: TYPE DITL                ;; Dialog item list
38:     ,1000 (32)           ;; Resource ID, (32)=purgeable
39: 7                          ;; Number of items following
40:
41: BtnItem Enabled          ;; 1. Ok button
42: 170 265 202 345
43: Ok
44:
45: RadioItem Enabled       ;; 2. Radio button #1
46: 25 30 41 130
47: White
48:
49: RadioItem Enabled       ;; 3. Radio button #2
50: 50 30 66 130
51: Black
52:
53: RadioItem Enabled       ;; 4. Radio button #3
54: 75 30 91 130
55: Gray
56:
57: RadioItem Enabled       ;; 5. Radio button #4
58: 100 30 116 130
59: LtGray
60:
61: RadioItem Enabled       ;; 6. Radio button #5
62: 125 30 141 130
63: DkGray
64:
65: StatText Disabled      ;; 7. Text
66: 175 30 196 195
67: Please select a pattern.
68:
69:
70: * END

```

Listing 6.8. RADIO.PAS

```

1: {$O Programs:Dialogs.F: }           { Send compiled code to here }
2: {$R Programs:Dialogs.F:Radio.Rsrc}  { Use this compiled resource file }
3: {$U-}                               { Turn off standard library units }
4:
5:
6: PROGRAM Radio;
7:
8: (*
9:
10: * PURPOSE : Radio buttons demo
11: * SYSTEM  : Macintosh / Turbo Pascal
12: * AUTHOR   : Tom Swan
13:
14: *)
15:
16:
17: {$U Programs:Units.F:DialogUnit }    { Open this library unit file }
18:
19:

```

(continued)

```

20:  USES
21:
22:      Mentypes, QuickDraw, OSIntf, ToolIntf, PackIntf, DialogUnit;
23:
24:
25:
26:  CONST
27:
28:      FileID          = 1;  { File menu Resource ID and commands }
29:      PatternsCmd      = 1;
30:      QuitCmd         = 2;
31:
32:      DialogID        = 1000; { Resource ID of radio buttons dialog }
33:
34:      WhiteButton     = 2;    { Button numbers corresponding to their }
35:      BlackButton     = 3;    { positions in the dialog's item list }
36:      GrayButton      = 4;
37:      LtGrayButton    = 5;
38:      DkGrayButton    = 6;
39:
40:
41:  VAR
42:
43:      fileMenu        : MenuHandle;    { Handle to program's only menu }
44:      theEvent         : EventRecord;   { Events from operating system }
45:      whichWindow      : WindowPtr;     { Window applying to event }
46:      quitRequested    : BOOLEAN;       { TRUE if quitting }
47:      pat              : Pattern;       { Fill pattern for dialog demo }
48:      buttons          : ButtonRec;     { Radio buttons info }
49:
50:
51:
52:  PROCEDURE DisplayPattern( dp : DialogPtr );
53:
54:  { Draw a filled box in the global pattern (pat) inside
55:    the dialog box at dp. }
56:
57:  VAR
58:
59:      r : Rect;
60:      oldPort : GrafPtr;
61:
62:  BEGIN
63:      IF buttons.selection <> -1 THEN
64:          BEGIN
65:              GetPort( oldPort );
66:              SetPort( dp );
67:              SetRect( r, 115, 25, 340, 147 );
68:              PenSize( 1, 1 );
69:              PenPat( black );
70:              FrameRect( r );
71:              InsetRect( r, 1, 1 );
72:              FillRect( r, pat );
73:              SetPort( oldPort )
74:          END { if }
75:      END; { DisplayPattern }
76:
77:
78:  PROCEDURE ChangePattern( dp : DialogPtr; buttons : ButtonRec );
79:
80:  { Change demonstration fill pattern according to current
81:    radio button selection }
82:

```

```

83: BEGIN
84:     CASE buttons.selection OF
85:         WhiteButton      : pat := white;
86:         BlackButton      : pat := black;
87:         GrayButton       : pat := gray;
88:         LtGrayButton     : pat := ltGray;
89:         DkGrayButton     : pat := dkGray;
90:     END; { case }
91:     DisplayPattern( dp )
92: END; { ChangePattern }
93:
94:
95: PROCEDURE DoPatterns;
96:
97: { Demonstrate using radios buttons to select a fill pattern.
98:   Also show how to display graphics inside a dialog box. }
99:
100: VAR
101:
102:     dp : DialogPtr;
103:     itemHit : INTEGER;
104:
105: BEGIN
106:     dp := GetNewDialog( DialogID, NIL, POINTER(-1) );
107:     IF dp <> NIL THEN
108:         BEGIN
109:             OutlineOk( dp );
110:             InitButtons( dp, buttons );
111:             ChangePattern( dp, buttons );
112:             REPEAT
113:                 ModalDialog( NIL, itemHit );
114:                 IF itemHit <> Ok THEN
115:                     BEGIN
116:                         PushButton( dp, buttons, itemHit );
117:                         ChangePattern( dp, buttons )
118:                     END { if }
119:                 UNTIL itemHit = Ok;
120:                 DisposDialog( dp )
121:             END { if }
122:         END; { DoPatterns }
123:
124:
125: PROCEDURE DoCommand( command : LongInt );
126:
127: { Execute a menu command }
128:
129: VAR
130:
131:     whichMenu      : INTEGER;      { Menu number of selected command }
132:     whichItem      : INTEGER;      { Menu item number of command }
133:
134: BEGIN
135:
136:     whichMenu := HiWord( command );      { Find the menu }
137:     whichItem := LoWord( command );      { Find the item }
138:
139:     IF whichMenu = FileID THEN
140:         CASE whichItem OF
141:             PatternsCmd : DoPatterns;
142:             QuitCmd      : quitRequested := TRUE
143:         END; { case }
144:         HiliteMenu( 0 ) { Unhighlight menu title }
145:
146:     END; { DoCommand }
147:
148:

```

(continued)

```

149: PROCEDURE MouseDownEvents;
150:
151: { Someone pressed the mouse button. Check its location and respond. }
152:
153:   VAR
154:
155:       partCode : INTEGER;      { Identifies what item was clicked. }
156:
157:   BEGIN
158:       WITH theEvent DO
159:           BEGIN
160:               partCode := FindWindow( where, whichWindow );
161:               CASE partCode OF
162:                   inMenuBar
163:                       : DoCommand( MenuSelect( where ) );
164:                   END { case }
165:               END { with }
166:           END; { MouseDownEvents }
167:
168:
169: PROCEDURE Initialize;
170:
171: { Program calls this routine one time at start }
172:
173:   BEGIN
174:
175:       InitGraf( @thePort );
176:       InitFonts;
177:       InitWindows;
178:       InitMenus;
179:       TEInit;
180:       InitDialogs( NIL );
181:       InitCursor;
182:       FlushEvents( everyEvent, 0 );
183:
184:       fileMenu := GetMenu( FileID );
185:       InsertMenu( fileMenu, 0 );
186:       DrawMenuBar;
187:
188:       quitRequested := FALSE;
189:
190:       WITH buttons DO
191:           BEGIN
192:               firstButton := WhiteButton;
193:               lastButton  := DkGrayButton;
194:               selection    := -1 { None }
195:           END { with }
196:
197:       END; { Initialize }
198:
199:
200: BEGIN
201:   Initialize;
202:   REPEAT
203:       SystemTask;
204:       IF GetNextEvent( everyEvent, theEvent ) THEN
205:           CASE theEvent.what OF
206:               MouseDown : MouseDownEvents;
207:           END { case }
208:       UNTIL quitRequested
209:   END.

```

Radio Play-by-Play

RADIO.R (1–70)

The resource text defines the dialog (23–30) for the test program. The item list (37–67) for this dialog contains seven objects: an Ok **BtnItem** (41–43), five **RadioItem** buttons (45–63) and a **StatText** message (65–67). Except for the word **RadioItem**, the definition of a radio button is the same as a **BtnItem**.

The text for a radio button appears to the right of the circle (see Figure 6.8). Many people don't realize that this text and the circle make a *single* object although they appear separately. In other words, you can click on the word Gray or inside the circle to select that button—you don't have to move the tip of the mouse pointer inside the tiny button circumference.

RADIO.PAS (1–209)

The program declares the resource ID for the dialog (32) and five constants (34–38). Each constant equals the position of one radio button in the dialog item list (see the resource text, Listing 6.7). Variable **buttons** (48) is a record of type **ButtonRec** with the structure in Figure 6.9.

Three integer fields in a **ButtonRec** variable hold the value of the first button (**firstButton**), the last button (**lastButton**), and the currently punched button (**selection**). The values of these fields are the same as the button item positions in the resource. In this example, **WhiteButton** is the first button and **DkGrayButton** the last. Lines 192–193 initialize the **buttons** record to these values. Setting the selection field to -1 (194) indicates that no button is punched. As you can see when you run the program, all buttons are white—none has a black center until you click one. You can change this feature by assigning to **selection** one of the five constants (34–38). For example, to preset the pattern to gray, change line 194 to read as follows and rerun the program:

```
selection := GrayButton;
```

Procedure **DisplayPattern** (52–75) draws a rectangle inside the dialog box and fills it with a pattern depending on the current value of **buttons.selection**. This

TYPE

```
ButtonRec =
  RECORD
    firstButton : INTEGER;    { First radio button item number }
    lastButton  : INTEGER;    { Last item number }
    selection   : INTEGER     { Current button (-1 if none) }
  END; { ButtonRec }
```

Figure 6.9 DialogUnit (Listing 6.18) defines this record data type to organize radio buttons.

demonstrates how to add features to dialog windows without defining those features as resources. To draw items in dialog windows, for example, use QuickDraw commands with a dialog pointer (**dp**) as you do with any other **grafPort** or window. (See lines 65–73.)

Procedure **ChangePattern** (78–92) sets global variable **pat** to one of QuickDraw's predefined patterns and then calls **DisplayPattern** (91) to draw the rectangle and fill it.

DoPatterns (95–122)

DoPatterns runs when you choose the Patterns command from the File menu. It first loads and displays the dialog window (106) and outlines the Ok button (109). Line 110 calls **InitButtons**, passing the dialog pointer **dp** and **buttons** record as parameters. **InitButtons** is a tool in DialogUnit. It sets the button controls according to the fields in the **ButtonRec** record, displaying any punched button with a black center. If you have more than one group of radio buttons in the same dialog—and there's no limit to the number of groups you can define—call **InitButtons** for each group.

Line 111 calls **ChangePattern** to display the rectangle and fill it with whatever **buttons.selection** indicates is the current pattern. This also initializes the global **pat** variable the first time you choose the Patterns command.

The program then repeatedly calls **ModalDialog** (113) until you click the Ok button. In this example, **ModalDialog** senses clicks inside radio buttons. If it returns a button value, line 116 calls DialogUnit tool **PushButton**, passing the dialog pointer **dp**, the **buttons** record, and the number of the item that **ModalDialog** returned in **itemHit**.

PushButton punches the new radio button, unpunches a previous button, and sets **buttons.selection** equal to **itemHit**, recording the current setting. If **itemHit** is not in the range of **buttons** fields **firstButton** to **lastButton**, **PushButton** does nothing. After handling the click in a radio button, the program calls **ChangePattern** again (117) to display the appropriate pattern.

SIMPLE DATA ENTRY

In conventional Pascal programming, to prompt for a name or number takes only two or three steps. For example, to ask someone their age, you might use the statements:

```
VAR age : INTEGER;

Write( 'What is your age? ' );
ReadLn( age );
```

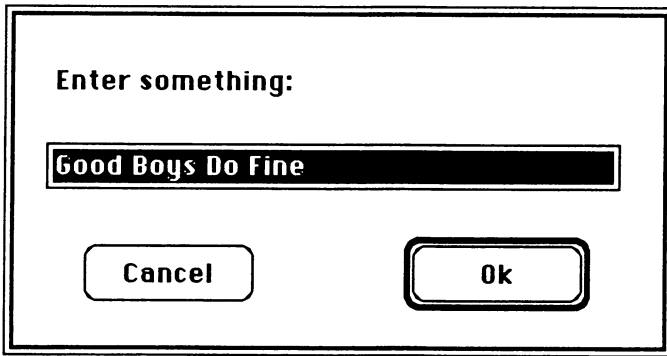


Figure 6.10 The dialog window of Listing 6.10, Entry.

On the Macintosh, you can do the same only with Turbo's textbook interface, which gives you a simulated terminal in which to ask such questions. In fully charged programs, you instead use a dialog with areas for typing these and other items.

The next example demonstrates how to use dialogs for simple data entry. This method activates standard editing features, double-clicking to select words, and backspacing to erase selected text. On newer systems with 128K ROMs, you also can use the four arrow keys to move the cursor within the editing area. Although the following example does not allow cutting and pasting text, it's possible to add those features as well, as the final program in this chapter demonstrates.

Type in Listing 6.9, save as ENTRY.R, and compile with RMaker. Type in Listing 6.10, save as ENTRY.PAS, and compile with Turbo to display the dialog window in Figure 6.10. Type whatever you like in the single-line edit area (darkened in the figure). Click Ok to save what you type so that it reappears when you again choose the File menu's Enter command. Click Cancel to throw out your typing, reverting to whatever you previously entered.

Listing 6.9. ENTRY.R

```

1: *-----*
2: * Entry.PAS resources -- Compile with RMaker      *
3: *-----*
4:
5: Programs:Dialogs.F:Entry.RSRC      ;; Send output to here
6:
7:
8: *-----*
9: * The File menu                                *
10: *-----*
11:
12: TYPE MENU
13:      ,1                                ;; Menu ID number to use in program
14: File                                ;; Menu title as shown in menu bar
15: Enter
16: Quit

```

(continued)

```

17:
18:
19: TYPE DLOG                                ;; Entry dialog box
20:     ,1000                                ;; Resource ID used in program
21: Entry                                    ;; Title (not displayed)
22: 100 100 250 400                          ;; Bounds rectangle top left bottom right
23: Visible NoGoAway                        ;; Visible on creation with no go-away box
24: 1                                        ;; Standard dialog style (dBoxProc)
25: 0                                        ;; RefCon value (none)
26: 1000                                    ;; Dialog IItem List (DITL) ID
27:
28:
29: TYPE DITL                                ;; Alert dialog item list
30:     ,1000 (32)                          ;; ID used in ALRT, (32)=purgeable
31: 4                                        ;; Number of items following
32: BtnItem Enabled                        ;; The Okay button
33: 105 185 132 265
34: Ok
35:
36: BtnItem Enabled                        ;; The Cancel button
37: 105 30 132 110
38: Cancel
39:
40: StatText Disabled                      ;; The prompting message
41: 20 15 36 145
42: Enter something:
43:
44: EditText Enabled                      ;; The entry text box
45: 60 15 76 280
46:
47:
48: * END

```

Listing 6.10. ENTRY.PAS

```

1: {$O Programs:Dialogs.F: }                { Send compiled code to here }
2: {$R Programs:Dialogs.F:Entry.Rsrc}      { Use this compiled resource file }
3: {$U-}                                    { Turn off standard library units }
4:
5:
6: PROGRAM Entry;
7:
8: (*
9:
10:  * PURPOSE : Simple data entry using modal dialog
11:  * SYSTEM  : Macintosh / Turbo Pascal
12:  * AUTHOR  : Tom Swan
13:
14: *)
15:
16:
17: {$U Programs:Units.F:DialogUnit }        { Open this library unit file }
18:
19:
20:     USES
21:
22:         Mementypes, QuickDraw, OSIntf, ToolIntf, PackIntf, DialogUnit;
23:
24:
25:

```

```

26:  CONST
27:
28:      FileID      = 1;      { Resource ID number for File menu }
29:      EnterCmd    = 1;      { Test Alert box text entry }
30:      QuitCmd     = 2;      { Quit command menu line number }
31:
32:      EntryID     = 1000;   { Resource ID of Entry dialog }
33:      EditItem    = 4;      { Position number of edit text item }
34:
35:
36:  VAR
37:
38:      fileMenu    : MenuHandle;   { Handle to program's only menu }
39:      message     : Str255;       { Sample entry string }
40:      theEvent    : EventRecord;   { Events from operating system }
41:      whichWindow : WindowPtr;     { Window applying to event }
42:      quitRequested : BOOLEAN;     { TRUE if quitting }
43:
44:
45:  PROCEDURE DoEntry( VAR message : Str255 );
46:
47:  { Demonstrate how to enter text in a dialog box }
48:
49:      VAR
50:
51:          dPtr      : DialogPtr;
52:          itemHit    : INTEGER;
53:          itemType   : INTEGER;
54:          itemRect   : Rect;
55:          itemHandle : Handle;
56:
57:  BEGIN
58:
59:      dPtr := GetNewDialog( EntryID,          { Resource ID }
60:                           NIL,              { Create template in heap }
61:                           POINTER(-1) );    { Make frontmost window }
62:
63:      IF dPtr <> NIL THEN
64:  BEGIN
65:      GetDItem( dPtr, EditItem, itemType, itemHandle, itemRect );
66:      IF itemHandle <> NIL THEN
67:  BEGIN
68:          OutLineOk( dPtr );
69:          SetIText( itemHandle, message );
70:          SelIText( dPtr, EditItem, 0, maxInt );
71:          REPEAT
72:              ModalDialog( NIL, itemHit )
73:          UNTIL ( itemHit = Ok ) OR ( itemHit = Cancel );
74:          IF itemHit = Ok
75:              THEN GetIText( itemHandle, message )
76:          END; { if }
77:          DisposDialog( dPtr ) { Dispose of dialog & related structures }
78:      END { if }
79:  END; { DoEntry }
80:
81:
82:  PROCEDURE DoFileMenuCommands( cmdNumber : INTEGER );
83:
84:  { Execute command in the File menu }
85:
86:  BEGIN
87:      CASE cmdNumber OF
88:          EnterCmd : DoEntry( message );
89:          QuitCmd  : quitRequested := TRUE
90:      END { case }
91:  END; { DoFileMenuCommands }
92:

```

(continued)

```

93:
94: PROCEDURE DoCommand( command : LongInt );
95:
96: { Execute a menu command }
97:
98:   VAR
99:
100:      whichMenu   : INTEGER;      { Menu number of selected command }
101:      whichItem   : INTEGER;      { Menu item number of command }
102:
103:   BEGIN
104:
105:      whichMenu := HiWord( command );      { Find the menu }
106:      whichItem := LoWord( command );      { Find the item }
107:
108:      CASE whichMenu OF
109:         FileID   : DoFileMenuCommands( whichItem );
110:      END; { case }
111:
112:      HiliteMenu( 0 ) { Unhighlight menu title }
113:
114:   END; { DoCommand }
115:
116:
117: PROCEDURE MouseDownEvents;
118:
119: { Someone pressed the mouse button. Check its location and respond. }
120:
121:   VAR
122:
123:      partCode : INTEGER;      { Identifies what item was clicked. }
124:
125:   BEGIN
126:      WITH theEvent DO
127:         BEGIN
128:            partCode := FindWindow( where, whichWindow );
129:            CASE partCode OF
130:               inMenuBar
131:                  : DoCommand( MenuSelect( where ) )
132:            END { case }
133:         END { with }
134:      END; { MouseDownEvents }
135:
136:
137: PROCEDURE Initialize;
138:
139: { Program calls this routine one time at start }
140:
141:   BEGIN
142:
143:      InitGraf( @thePort );
144:      InitFonts;
145:      InitWindows;
146:      InitMenus;
147:      TEInit;
148:      InitDialogs( NIL );
149:      InitCursor;
150:      FlushEvents( everyEvent, 0 );
151:
152:      fileMenu := GetMenu( FileID );
153:      InsertMenu( fileMenu, 0 );
154:      DrawMenuBar;
155:
156:      quitRequested := FALSE;
157:
158:      message := ''
159:
160:   END; { Initialize }

```

```
161:
162:
163: BEGIN
164:   Initialize;
165:   REPEAT
166:     SystemTask;
167:     IF GetNextEvent( everyEvent, theEvent ) THEN
168:       CASE theEvent.what OF
169:        MouseDown : MouseDownEvents;
170:       END { case }
171:     UNTIL quitRequested
172:   END.
```

Entry Play-by-Play

ENTRY.R (1-48)

The resource text declares a single dialog with ID 1000 (19-26). Of the four objects in the item list (29-45), two are buttons, one is the prompting message, and the last is the editing area, of type **Edit Text** (44-45). Notice that **Edit Text** items have only two lines instead of three as do most other objects in an item list.

As line 45 shows, the edit area is 16 pixels tall, the minimum height you can use with the standard text font. This value exactly centers the darkened text inside the top and bottom borders of the editing box (Figure 6.10), which the dialog draws for you.

ENTRY.PAS (1-172)

Two constants hold the dialog’s resource ID, **EntryID** (32), and the position of the **EditText** object in the dialog’s item list, **EditItem** (33). A global variable, **message** (39), holds whatever you type in the edit box after you click Ok.

Procedure **DoEntry** (45-79) handles the entire dialog. It loads and displays the dialog resource in the usual way (59-61) and then calls **GetDItem** (65) passing the dialog pointer (**dPtr**) and the appropriate item number (**EditItem**). **GetDItem** finds this item in the dialog’s item list now in memory and returns the remaining three parameters. **ItemType** indicates the item kind according to the list in Table 6.2. Although this example ignores **ItemType**, you can use it to check whether **Get-**

Table 6.2 Dialog item list types.

Name	ItemType	Constant
Button	0	BtnCtrl
Check box	1	ChkCtrl
Radio button	2	RadCtrl
Static text	8	StatText
Edit box	16	EditText
Icon	32	IconItem
Picture	64	QuickDraw picture

DItem finds the type of item you expect. **ItemHandle** is a handle to the item on the heap. The final parameter, **itemRect**, is the item's enclosing rectangle in coordinates local to the dialog window.

As long as **itemHandle** is not **NIL**, the statements at lines 68–75 first outline the Ok button and then call two procedures that you'll almost always use with edit text items in dialogs. **SetItemText** (Set Item Text) (69) takes two parameters: the item handle from **GetDItem** and a string, **message** in the example. It inserts the string into the dialog item, copying its characters for editing. Because of that, the actual string variable (**message**) never changes.

In line 70, **SelectItemText** (Select Item Text) takes four parameters, the dialog pointer (**dPtr**), a constant representing the item's position in the dialog item list (**EditItem**), and two integer values, **0** and **MaxInt** in the example. These two values represent the range of character positions you want the dialog to highlight (darken) inside the edit box (see Figure 6.10). Setting the first integer to **0** and the second to **MaxInt** (Maximum Integer) selects all text in the box unless the string parameter in the previous statement (69) was empty.

The **REPEAT** loop (71–73) is identical to previous examples. It calls **ModalDialog** until **itemHit** equals Ok or Cancel, indicating a mouse click in one of those two buttons. If it equals Ok, then line 75 calls **GetItemText** (Get Item Text) passing the item handle and a string variable. This copies the edited text back into the string, ready for editing the next time you open this same dialog.

To initialize a string to a default value, assign any string to **message** at line 158. Remember that editing text in dialogs does not create places to store characters in memory. It's up to you to keep string variables like **message** and then pass those variables to **SetItemText**, retrieving them after editing with **GetItemText** as the example demonstrates.

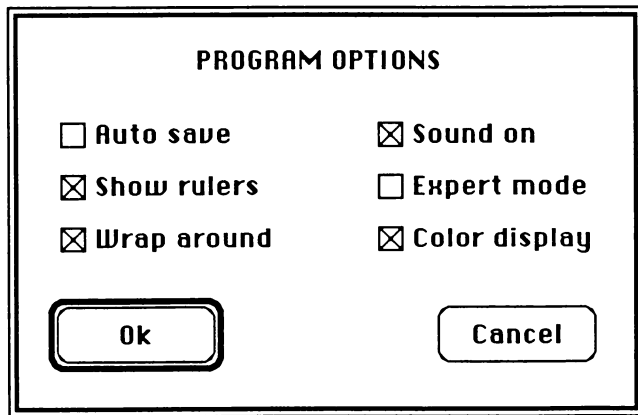


Figure 6.11 The dialog window of Listing 6.12. Options.

CHECK BOXES

One of the most common uses for check boxes in dialogs is to select among various program options. DialogUnit (Listing 6.18) contains data types and routines for adding this feature to programs.

Figure 6.11 shows the dialog that the next example displays. The options are imaginary and don't cause any changes—feel free to experiment. Type in Listing 6.11, save as OPTIONS.R and compile with RMaker to produce the program's resource file. Next, type in Listing 6.12 and save as OPTIONS.PAS. Compile with Turbo to run the test.

Listing 6.11. OPTIONS.R

```

1: *-----*
2: * Options.PAS resources -- Compile with RMaker *
3: *-----*
4:
5: Programs:Dialogs.F:Options.RSRC      ;; Send output to here
6:
7:
8: *-----*
9: * The File menu *
10: *-----*
11:
12: TYPE MENU
13:     ,1
14: File
15:     Options
16:     Quit
17:
18:
19: *-----*
20: * The options dialog *
21: *-----*
22:
23: type DLOG
24:     ,1000                ;; Resource ID
25: Options                 ;; Title (not displayed)
26: 90 110 270 395          ;; Top, Left, Bottom, Right
27: Visible NoGoAway        ;; Immediately visible, no go-away box
28: 1                       ;; Std double-border dialog window
29: 0                       ;; Reference value (none)
30: 1000                    ;; ID of dialog item list (following)
31:
32:
33: *-----*
34: * The options dialog item list *
35: *-----*
36:
37: TYPE DITL                ;; Dialog item list
38:     ,1000 (32)           ;; Resource ID, (32)=purgeable
39: 9                        ;; Number of items following
40:
41: BtnItem Enabled          ;; 1. Ok button
42: 135 15 162 90
43: Ok
44:

```

(continued)


```

45: BtnItem Enabled           ;; 2. Cancel button
46: 135 195 162 270
47: Cancel
48:
49: ChkItem Enabled           ;; 3. Option #1
50: 45 15 61 100
51: Auto save
52:
53: ChkItem Enabled           ;; 4. Option #2
54: 70 15 86 115
55: Show rulers
56:
57: ChkItem Enabled           ;; 5. Option #3
58: 95 15 111 120
59: Wrap around
60:
61: ChkItem Enabled           ;; 6. Option #4
62: 45 165 61 250
63: Sound on
64:
65: ChkItem Enabled           ;; 7. Option #5
66: 70 165 86 270
67: Expert mode
68:
69: ChkItem Enabled           ;; 8. Option #6
70: 95 165 111 270
71: Color display
72:
73: StatText Disabled         ;; 9. Text
74: 10 80 26 200
75: PROGRAM OPTIONS
76:
77:
78: * END

```

Listing 6.12. OPTIONS.PAS

```

1: {$O Programs:Dialogs.F: }           { Send compiled code to here }
2: {$R Programs:Dialogs.F:Options.Rsrc} { Use this compiled resource file }
3: {$U-}                                { Turn off standard library units }
4:
5:
6: PROGRAM Options;
7:
8: (*
9:
10:  * PURPOSE : Program option selector demo
11:  * SYSTEM  : Macintosh / Turbo Pascal
12:  * AUTHOR   : Tom Swan
13:
14:  *)
15:
16:
17: {$U Programs:Units.F:DialogUnit }    { Open this library unit file }
18:
19:
20:  USES
21:
22:      Memtypes, QuickDraw, OSIntf, ToolIntf, PackIntf, DialogUnit;
23:
24:

```

```

25:
26:  CONST
27:
28:      FileID          = 1;  { File menu Resource ID and commands }
29:      OptionsCmd      = 1;
30:      QuitCmd         = 2;
31:
32:      DialogID        = 1000; { Resource ID of radio buttons dialog }
33:
34:      OptAutoSave     = 3;    { Check box numbers corresponding }
35:      OptShowRulers   = 4;    { to their positions in the dialog }
36:      OptWrapAround   = 5;    { item list. }
37:      OptSound        = 6;
38:      OptExpertMode   = 7;
39:      OptColor        = 8;
40:
41:
42:  VAR
43:
44:      fileMenu        : MenuHandle;    { Handle to program's only menu }
45:      theEvent        : EventRecord;    { Events from operating system }
46:      whichWindow      : WindowPtr;     { Window applying to event }
47:      quitRequested   : BOOLEAN;        { TRUE if quitting }
48:      programOptions  : ChecksRecord;   { Program options information }
49:
50:
51:
52:  PROCEDURE DoOptions;
53:
54:  { Demonstrate how to prompt for various program options using
55:    check boxes in a dialog window }
56:
57:  VAR
58:
59:      dp : DialogPtr;
60:      itemHit : INTEGER;
61:      checks : ChecksRecord;
62:
63:  BEGIN
64:      checks := programOptions;
65:      dp := GetNewDialog( DialogID, NIL, Pointer(-1) );
66:      IF dp <> NIL THEN
67:          BEGIN
68:              OutlineOk( dp );
69:              InitChecks( dp, checks );
70:              REPEAT
71:                  ModalDialog( NIL, itemHit );
72:                  IF ( itemHit <> Ok ) AND ( itemHit <> Cancel )
73:                      THEN CheckBox( dp, checks, itemHit )
74:                  UNTIL ( itemHit = Ok ) OR ( itemHit = Cancel );
75:                  IF itemHit = Ok
76:                      THEN programOptions := checks;
77:                  DisposDialog( dp )
78:              END { if }
79:          END; { DoOptions }
80:
81:
82:  PROCEDURE DoCommand( command : LongInt );
83:
84:  { Execute a menu command }
85:
86:  VAR
87:
88:      whichMenu      : INTEGER;    { Menu number of selected command }
89:      whichItem      : INTEGER;    { Menu item number of command }
90:

```

(continued)

```

91:   BEGIN
92:
93:       whichMenu := HiWord( command );      { Find the menu }
94:       whichItem := LoWord( command );      { Find the item }
95:
96:       IF whichMenu = FileID THEN
97:           CASE whichItem OF
98:               OptionsCmd : DoOptions;
99:               QuitCmd    : quitRequested := TRUE
100:            END; { case }
101:       HiliteMenu( 0 ) { Unhighlight menu title }
102:
103:   END; { DoCommand }
104:
105:
106: PROCEDURE MouseDownEvents;
107:
108: { Someone pressed the mouse button. Check its location and respond. }
109:
110:   VAR
111:
112:       partCode : INTEGER;      { Identifies what item was clicked. }
113:
114:   BEGIN
115:       WITH theEvent DO
116:           BEGIN
117:               partCode := FindWindow( where, whichWindow );
118:               CASE partCode OF
119:                   inMenuBar
120:                       : DoCommand( MenuSelect( where ) );
121:               END { case }
122:           END { with }
123:       END; { MouseDownEvents }
124:
125:
126: PROCEDURE Initialize;
127:
128: { Program calls this routine one time at start }
129:
130:   BEGIN
131:
132:       InitGraf( @thePort );
133:       InitFonts;
134:       InitWindows;
135:       InitMenus;
136:       TEInit;
137:       InitDialogs( NIL );
138:       InitCursor;
139:       FlushEvents( everyEvent, 0 );
140:
141:       fileMenu := GetMenu( FileID );
142:       InsertMenu( fileMenu, 0 );
143:       DrawMenuBar;
144:
145:       quitRequested := FALSE;
146:
147:       WITH programOptions DO
148:           BEGIN
149:               firstCheck := OptAutoSave;
150:               lastCheck  := OptColor;
151:
152:               { Set up default options }
153:
154:               selections :=
155:                   [ OptShowRulers, OptWrapAround, OptSound, OptColor ]
156:
157:           END { with }
158:
159:   END; { Initialize }

```

```

160:
161:
162: BEGIN
163:   Initialize;
164:   REPEAT
165:     SystemTask;
166:     IF GetNextEvent( everyEvent, theEvent ) THEN
167:       CASE theEvent.what OF
168:        MouseDown    : MouseDownEvents;
169:       END { case }
170:     UNTIL quitRequested
171:   END.

```

Options Play-by-Play

OPTIONS.R (1-78)

The resource text file resembles previous examples. The dialog's item list (37-75) defines nine items including two **BtnItem** buttons (41-47), six **ChkItem** check boxes (49-71), and a **StatText** item (73-75) for the title at the top of the window (see Figure 6.11).

Notice that you define only the check box locations and labels. You do not specify whether a box has a check mark inside. As the next section explains, that's the program's responsibility.

OPTIONS.PAS (1-171)

The program declares six constants (34-39) equal to the **ChkItem** positions in the dialog's item list, in this example, 3 to 8. Line 48 declares record **programOptions** as type **ChecksRecord**, a data type from **DialogUnit**.

Figure 6.12 describes the fields in this record. A **CheckSet** is a set of any values from zero to 255, the range you can use for check boxes. In other words, you can have up to 255 check boxes in a single dialog. The **ChecksRecord** type holds two integer fields, **firstCheck** and **lastCheck**. As in the **ButtonRec** type (see Figure 6.9), these fields specify the range of item numbers in the dialog. In this example,

```

TYPE

CheckSet = SET OF 0 .. 255;      { Sets of checked boxes }

ChecksRecord =
  RECORD
    firstCheck  : INTEGER;      { First item number in check list }
    lastCheck   : INTEGER;      { Last item number }
    selections  : CheckSet      { Set of currently checked boxes }
  END; { ChecksRecord }

```

Figure 6.12 DialogUnit (Listing 6.18) defines these record and set data types to organize check boxes.

firstCheck is 3, the value of constant **OptAutoSave** (34), and **lastCheck** is 8, the value of **OptColor** (39). The **selections** field in **ChecksRecord** is the set of all check boxes currently on.

Lines 147–157 initialize a **ChecksRecord** variable, **programOptions** in this example. After setting **firstCheck** and **lastCheck** (149–150), the program assigns the set of default options to **selections** (154–155). To select a different set of defaults, change the elements in brackets at line 155 to include any of the six constants (34–39). For example, to turn on the auto-save and sound options, change 154–155 to the single line:

```
selections := [ OptAutoSave, OptSound ];
```

Procedure **DoOptions** (52–79) runs when you choose the File menu's Options command. Line 64 makes a temporary copy of the global **programOptions** record. The procedure then lets you change the options in this copy, assigning it back to the global record only if you click the Ok button. Clicking Cancel throws away any changes you make.

Line 65 loads and displays the dialog in the usual way and line 68 outlines the Ok button. **InitChecks** (69) is similar to **InitButtons** (see Listing 6.8, 110). Pass a dialog pointer and **ChecksRecord** variable to preset check boxes to the current selections set, adding visible checks to all boxes in that set.

A **REPEAT** loop then calls **ModalDialog** (71), which returns in integer **itemHit** the number of the item clicked. If that item is not one of the dialog's two buttons, line 73 calls DialogUnit tool **CheckBox** to toggle a check mark on or off. In your own programs, pass the dialog pointer, **ChecksRecord** variable and **itemHit** as in the example. **CheckBox** displays visible checks where necessary and updates the **selection** field in the **checks** record.

Notice that lines 75–76 assign the temporary **checks** copy back to global **programOptions** only if you click the Ok button. After that, line 77 erases the dialog from memory before the procedure ends.

Using Options in Programs

As you can see from Listing 6.12, the check boxes in the options dialog don't cause any actions to occur. They merely change the settings the program stores in its **ChecksRecord** variable. But the example doesn't show how to make use of those options—something a real program would have to do.

For an example, follow these steps to activate the Sound option in the example. With Sound on, you hear a beep every time you click the mouse (except when checking boxes in the options dialog itself.) Add these lines between lines 114 and 115 in procedure **MouseDownEvents**.

```
IF OptSound IN programOptions.selections
  THEN SysBeep( 3 );
```

Run the modified program and click the mouse anywhere in the desktop. You should hear a beep. Choose the Options command from the File menu and turn off the Sound option. After clicking Ok, you no longer hear beeps when you click the mouse.

As you can see, the dialog itself does not keep track of the program options—it merely displays a visual representation of whatever settings you tell it about. It's your responsibility to save those settings and respond appropriately as this example shows.

To test for various options, use standard Pascal set commands. When you want to test for a single option, use the **IN** operator like this:

```
IF optionConstant IN programOptions.selections
  THEN { option-on action }
  ELSE { option-off action }
```

To test for option groups, use other Pascal set operators. For example, to check whether the auto-save and color options are on in Listing 6.12, you could write:

```
IF [ OptSound, OptColor ] <=
  programOptions.selections
  THEN SysBeep( 3 );
```

In other words, only if the set of **OptSound** and **OptColor** is included (**<=**) in the **programOptions** selections set does the program beep. Try this statement in place of the earlier modification you made between lines 114 and 115. With this change, you now have to check both the sound and color options to make mouse clicks beep.

ERROR MESSAGES

Designing error messages as dialogs or alerts gives them a consistent look and makes it easy to add new error messages to programs as you write them. The next example develops an error message center that you can use in any program. Included is a procedure for displaying Turbo's **IoResult** error codes for use with **Reset** and **Rewrite** (along with other procedures) when you turn off I/O error checking with the **{Si-}** option. The same program can test other error messages, too.

The first part of the error system is a unit of tools that help add error alerts to programs. Type in Listing 6.13 and save as **ERRORUNIT.PAS**. Compile to a disk file in folder **Units** or change line 1 to compile to a different volume and folder. **ErrorUnit** uses **DialogUnit** (Listing 6.18), which it expects to find in the volume and folder in line 19. After compiling, don't try to run **ErrorUnit**—you need to write a host program to use its routines.

Listing 6.13. ERRORUNIT.PAS

```

1: {$O Programs:Units.F: }      { Send compiled code to here }
2: {$U-}                        { Turn off standard library units }
3:
4:
5: UNIT ErrorUnit( 131 );
6:
7: (*
8:
9:  * PURPOSE : Error message alert tools
10:  * SYSTEM  : Macintosh / Turbo Pascal
11:  * AUTHOR   : Tom Swan
12:
13:  *)
14:
15:
16: INTERFACE                      { Items visible to a host program }
17:
18:
19: {$U Programs:Units.F:DialogUnit } { Open this library unit file }
20:
21:
22:  USES
23:
24:      Memtypes, QuickDraw, OSIntf, ToolIntf, PackIntf, DialogUnit;
25:
26:
27:  CONST
28:
29:      ErrorID          = 999;    { Error alert box resource ID }
30:
31:
32:  TYPE
33:
34:      ErrorType = ( StopError, NoteError, CautionError );
35:
36:
37: PROCEDURE DisplayError( errNum : INTEGER; errMessage,
38:                        errHelp : Str255; errKind : ErrorType );
39:
40: PROCEDURE IOError( errNum : INTEGER; helpMessage : Str255 );
41:
42:
43:
44: IMPLEMENTATION                  { Items not visible to a host program }
45:
46:
47: PROCEDURE DisplayError;
48:
49: { Display error number, message, and help strings.  }
50:
51:  VAR
52:
53:      errNumStr    : Str255;
54:      itemHit      : INTEGER;
55:
56:  BEGIN
57:
58:      NumToString( errNum, errNumStr );
59:      ParamText( errNumStr, errMessage, errHelp, '' );
60:      CASE errKind OF
61:          StopError      : itemHit := StopAlert( ErrorID, NIL );
62:          NoteError      : itemHit := NoteAlert( ErrorID, NIL );
63:          CautionError  : itemHit := CautionAlert( ErrorID, NIL );
64:      END { case }
65:
66:  END; { DisplayError }

```

```

67:
68:
69: PROCEDURE IOError;
70:
71: { Display message for this standard IOResult error number }
72:
73:   VAR
74:
75:       s : Str255;
76:
77:   BEGIN
78:
79:       CASE errNum OF
80:           -33 : s := 'File directory full';
81:           -34 : s := 'Volume allocation blocks full';
82:           -35 : s := 'Volume does not exist';
83:           -36 : s := 'Disk I/O error';
84:           -37 : s := 'Bad file or volume name';
85:           -38 : s := 'File not open';
86:           -39 : s := 'Unexpected end of file';
87:           -40 : s := 'Reference before start of file';
88:           -41 : s := 'System heap is full';
89:           -42 : s := 'Too many open files';
90:           -43 : s := 'File not found';
91:           -44 : s := 'Write protect tab open';
92:           -45 : s := 'File is locked';
93:           -46 : s := 'Volume locked';
94:           -47 : s := 'One or more files open';
95:           -48 : s := 'File already exists';
96:           -49 : s := 'Attempt to write to already open file';
97:           -50 : s := 'No default volume';
98:           -51 : s := 'Bad file reference number';
99:           -53 : s := 'Volume not on line';
100:          -54, -61 : s := 'Writing not permitted';
101:          -55 : s := 'Volume already mounted and on line';
102:          -56 : s := 'Bad drive number';
103:          -57 : s := 'Not a Macintosh format directory';
104:          -58 : s := 'Problem in external file system';
105:          -59 : s := 'Cannot rename';
106:          -60 : s := 'Bad master directory block';
107:          -108 : s := 'Heap zone full';
108:          -120 : s := 'Directory not found';
109:          -121 : s := 'Too many directories open';
110:          -122 : s := 'Bad HFS command';
111:          -123 : s := 'Non HFS-directory';
112:          -127 : s := 'Internal file system error';
113:          -128 : s := 'Text file not open for input';
114:          -129 : s := 'Text file not open for output';
115:          -130 : s := 'Error in number'
116:
117:       OTHERWISE
118:
119:           BEGIN
120:               s := 'Unknown error condition';
121:               helpMessage :=
122:                   'Please notify programmer at (800) 555-1212'
123:           END
124:
125:       END; { case }
126:
127:       DisplayError( errNum, s, helpMessage, StopError )
128:
129:   END; { IOError }
130:
131:
132:
133: END. { ErrorUnit }

```


ErrorUnit Play-by-Play

Programs that use ErrorUnit must also use all the units line 24 lists. You also need an alert dialog with resource ID 999, which the unit declares as constant **ErrorID** at line 29. Following this section is an example that shows how to design the alert.

ErrorType (34) is an enumerated data type with three elements, **StopError**, **NoteError**, and **CautionError**. Pass one of these elements to procedure **DisplayError** to select one of the alert icons in Figure 6.7.

```
PROCEDURE DisplayError( errNum : INTEGER;
  errMessage, errHelp : Str255;
  errKind : ErrorType );
```

DisplayError (47–66) takes four parameters: an error number (**errNum**), a string (**errMessage**), an optional help line (**errHelp**), and the error type (**errKind**). Although the two strings **errMessage** and **errHelp** are of type **Str255**, they can never be as long as that. Because the Macintosh uses proportionally spaced characters, it's impossible to give a maximum string length although 40 characters each should be a reasonable limit. Use the program in the next section to test your messages to make certain none is too long.

ErrNum can be any integer value from –32768 to 32767. **ErrMessage** should be a description of what that error code means. **ErrHelp** should offer advice and consolation. And **errKind** should specify an appropriate icon from Figure 6.7 for the upper left corner of the alert window. (Hint: reserve the stop alert's exclamation for serious problems. Don't overdo your exclamation points!!!)

For example, **IoResult** error code –43 indicates that **Reset** could not find the file name you told it to open. To display the error message in Figure 6.13, you could write:

```
{ $i- } Reset( f, fileName ); { $i+ }
errCode := IoResult;
IF errCode <> 0 THEN
  DisplayError( errCode, 'File not found',
    'Check disk and try operation again', StopError );
```

Even better is to pass the file name as part of the message. Rather than display only a note that the program failed to find a file, this tells people which file it couldn't find, a fact that might not be obvious. To do that, call **DisplayError** with a **Concat** statement to add the file name to a message.

```
DisplayError( errCode, 'File not found',
  Concat('Did you remember to create ', fileName, '?'),
  StopError );
```

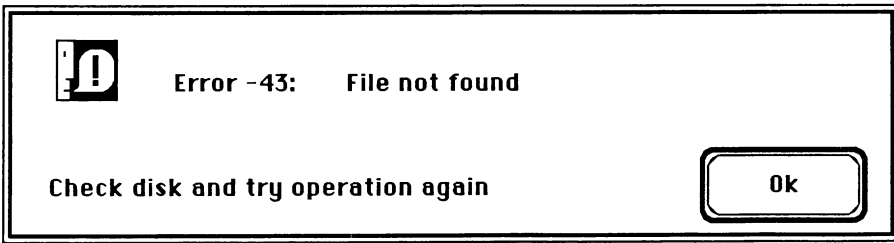


Figure 6.13 ErrorUnit (Listing 6.13) makes it easy to display error messages in alert windows as this example shows.

As Figure 6.13 shows, the error number follows the word “Error” to the right of the icon. After that comes the error message. The help message appears along the bottom of the window. If you don’t want either message to appear, pass a null string (‘’) for either **errMessage** or **errHelp**.

DisplayError converts the error code to a string (58) and passes it along with the error and help messages to **ParamText** (59). It then calls one of the three alert functions (60–64) depending on **errKind**’s value. Because there’s only one button, it ignores **itemHit**, the value that alert functions return.

```
PROCEDURE IOError( errNum : INTEGER;
  helpMessage : Str255 );
```

The second procedure in ErrorUnit (69–129) displays a message corresponding to the error codes that **IoResult** returns. Instead of the previous suggestions, use the procedure this way:

```
{ $r- } Reset( f, fileName ); { $r+ }
errCode := IoResult;
IF errCode <> 0
  THEN IOError( errCode, 'Check disk drives' )
  ELSE { continue with program }
```

Of course you can pass whatever help message you want as the string parameter to **IOError**. Although the procedure knows all the error codes listed in the Turbo manual, it’s possible that future revisions of the compiler will add new error numbers. For that reason, if **IOError** receives an unknown code, it displays the note: “Please notify programmer at (800) 555-1212.” You’ll probably want to change line 122 to display something else here.

IOError breaks a Macintosh rule that English language strings should be in resource files instead of directly encoded in programs. That’s probably a good idea for finished programs but, in this case, because you have the program source text, it’s just as easy to translate the strings in the procedure as it is to modify compiled


```

12: TYPE MENU                      ;; Dummy menu used as title
13:     ,1
14: Error Test -- Type 0 to quit
15:
16:
17: *-----*
18: * The error number entry dialog *
19: *-----*
20:
21: TYPE DLOG
22:     ,1000                      ;; Resource ID
23: Test Entry                      ;; Title (not displayed)
24: 125 150 232 357                ;; Top Left Bottom Right
25: Visible NoGoAway                ;; Immediately visible, no go-away box
26: 1                              ;; Standard double-border dialog window
27: 0                              ;; Reference value (none)
28: 1000                          ;; Item list ID (following)
29:
30:
31: TYPE DITL                      ;; Error number dialog item list
32:     ,1000 (32)                 ;; Resource ID, (32)=purgeable
33: 3                              ;; Number of items following
34:
35: BtnItem Enabled                 ;; 1. OK button
36: 70 63 95 142
37: Ok
38:
39: EditText Enabled                ;; 2. Box to enter error number
40: 25 140 41 175
41: 0
42:
43: StatText Disabled              ;; 3. Prompt
44: 25 25 41 125
45: Error number?
46:
47:
48: *-----*
49: * The error alert template *
50: *-----*
51:
52: TYPE ALRT
53:     ,999 (4)                   ;; Resource ID, (4) = preload
54: 45 28 154 481                 ;; Top Left Bottom Right
55: 999                          ;; Item list ID (following)
56: 5555                          ;; Stages (none)
57:
58:
59: TYPE DITL                      ;; Error alert item list
60:     ,999 (32)                 ;; Resource ID , (32)=purgeable
61: 4
62:
63: BtnItem Enabled                 ;; 1. Ok button
64: 71 360 103 440
65: Ok
66:
67: StatText Disabled              ;; 2. Error number
68: 25 80 41 165
69: Error ^0:
70:
71: StatText Disabled              ;; 3. Error message
72: 25 170 41 440
73: ^1
74:
75: StatText Disabled              ;; 4. Help message
76: 80 15 96 350
77: ^2
78:
79:
80: * END

```

Listing 6.15. ERRTEST.PAS

```

1: {$O Programs:Dialogs.F: }           { Send compiled code to here }
2: {$R Programs:Dialogs.F:ErrTest.Rsrc} { Use this compiled resource file }
3: {$U-}                               { Turn off standard library units }
4:
5:
6: PROGRAM ErrTest;
7:
8: (*
9:
10:  * PURPOSE : Test error messages
11:  * SYSTEM  : Macintosh / Turbo Pascal
12:  * AUTHOR  : Tom Swan
13:
14: *)
15:
16:
17: {$U Programs:Units.F:DialogUnit }    { Open these library unit files }
18: {$U Programs:Units.F:ErrorUnit }
19:
20:
21: USES
22:
23:     Memtypes, QuickDraw, OSIntf, ToolIntf, PackIntf,
24:     DialogUnit, ErrorUnit;
25:
26:
27:
28: CONST
29:
30:     TitleID          = 1;      { Resource ID for dummy Title menu }
31:     NumEntryID       = 1000;   { Error number entry dialog resource ID }
32:
33:
34: VAR
35:
36:     titleMenu        : MenuHandle; { Handle to program's only menu }
37:     quitRequested    : BOOLEAN;    { TRUE if quitting }
38:     errNum           : INTEGER;    { Test error number }
39:
40:
41:
42: PROCEDURE GetErrorNumber( VAR errNum : INTEGER );
43:
44: { Prompt for and return an integer error number }
45:
46: CONST
47:
48:     EditItem        = 2;      { Dialog item number in NumEntryID resource }
49:
50: VAR
51:
52:     dPtr            : DialogPtr;
53:     itemHit         : INTEGER;
54:     itemType        : INTEGER;
55:     itemRect        : Rect;
56:     itemHandle       : Handle;
57:     s                : Str255;
58:     temp            : LONGINT;
59:
60: BEGIN
61:
62:     dPtr := GetNewDialog( NumEntryID, NIL, POINTER(-1) );
63:

```

```

64:      IF dPtr <> NIL THEN
65:      BEGIN
66:          GetDItem( dPtr, EditItem, itemType, itemHandle, itemRect );
67:          IF itemHandle <> NIL THEN
68:          BEGIN
69:              OutLineOk( dPtr );
70:              NumToString( errNum, s );
71:              SetIText( itemHandle, s );
72:              SelIText( dPtr, EditItem, 0, maxInt );
73:              REPEAT
74:                  ModalDialog( NIL, itemHit )
75:              UNTIL itemHit = Ok;
76:              GetIText( itemHandle, s );
77:              StringToNum( s, temp );
78:              errNum := temp
79:          END; { if }
80:          DisposDialog( dPtr )
81:      END { if }
82:  END; { GetErrorNumber }
83:
84:
85: PROCEDURE TestError( errNum : INTEGER );
86:
87: { Call IOError with appropriate help message }
88:
89: BEGIN
90:     IF errNum <> 0
91:     THEN IOError( errNum, 'Check disk and try operation again' )
92:     END; { TestError }
93:
94:
95: PROCEDURE DoTest;
96:
97: { Type in an error number and display error message }
98:
99: BEGIN
100:     GetErrorNumber( errNum );
101:     TestError( errNum )
102: END; { DoTest }
103:
104:
105: PROCEDURE Initialize;
106:
107: { Program calls this routine one time at start }
108:
109: BEGIN
110:
111:     InitGraf( @thePort );
112:     InitFonts;
113:     InitWindows;
114:     InitMenus;
115:     TEInit;
116:     InitDialogs( NIL );
117:     InitCursor;
118:     FlushEvents( everyEvent, 0 );
119:
120:     titleMenu := GetMenu( TitleID ); { Display dummy menu }
121:     InsertMenu( titleMenu, 0 );
122:     DrawMenuBar;
123:
124:     errNum := 0
125:
126: END; { Initialize }
127:
128:

```

(continued)

```

129: BEGIN
130:   Initialize;
131:   REPEAT
132:     SystemTask;
133:     DoTest
134:   UNTIL errNum = 0
135: END.

```

ErrTest Play-by-Play

ERRTEST.R (1-80)

The resource text file declares a dialog template (21-28) for the error number entry window in Figure 6.14. The item list (31-45) contains three objects: an Ok button, an **Edit Text** box for typing in numbers, and a **Stat Text** prompt.

The error alert (52-56) has ID 999, which ErrorUnit requires. In your own programs, copy this alert template exactly as listed. Its item list (59-77) has an Ok button and three **Stat Text** items with replaceable parameters. Parameter ^0 is the error number, ^1 is the error message, and ^2 is the help line. These symbols correspond with the strings you pass to **ParamText**. (See Listing 6.13, 59.)

Most unusual is the menu declaration at lines 12-14. The program displays this menu title as a prompt—it has no pull-down menus. Although this use of a menu resource would be unacceptable in a commercial program, it's a useful trick to remember when writing tests and demos like this one.

ERRTEST.PAS (1-135)

Because the test program has no pull-down menus, its main loop (129-135) differs from other examples in this book. After initializing, a **REPEAT** loop calls **SystemTask** and **DoTest** until global **errNum** is zero. (Calling **SystemTask** might not be necessary—the program can't activate desk accessories. But in the Mac's world of program switchers, networks, and print spoolers, it's safer to call this procedure often rather than risk starving some poor process in memory waiting for its hand-out of time.)

Procedure **DoTest** (95-102) calls **GetErrorNumber** and **TestError** in succession. **GetErrorNumber** (42-82) displays the dialog in Figure 6.14. After loading and displaying the dialog window (62), it calls **GetDItem** (66) to set **itemHandle** to the **Edit Text** item for typing error codes (the darkened box in the figure). It then converts the global **errNum** to a string (70) and calls **SetIText** and **SellText** (71-72) to change the number in the box to **errNum**'s value and select it for editing.

The **REPEAT** loop (73-75) calls **ModalDialog**, just as in other dialog examples, letting you type in new numbers until you click Ok. After that, the procedure converts the **Edit Text** item back into a **LONGINT** value (77) and assigns it to **errNum**. (**StringToNum** requires a **LONGINT** parameter—the reason for using a temporary variable here rather than directly passing integer **errNum** as a parameter.)

Procedure **TestError** (85–92) calls **ErrorUnit** routine **IOError**, passing the error number and string as parameters. To test your own errors, call your error procedure here instead of **IOError**.

DATA ENTRY FORMS

The next example, one of the longest in this book, combines many features of previous chapters with dialogs to make a data entry system that you can use to enter records of various kinds. Figure 6.15 is a copy of the program's display, a nine-field entry form for a name and address database.

At the same time, the example shows how to program modeless dialogs, allowing you to activate pull-down menus and desk accessories. In this way, the dialog is more like a regular program window but, instead of having to design the window's contents yourself, you call dialog procedures to edit text, display controls, and use other dialog features demonstrated earlier.

As with all examples in this chapter, this one uses the programming in **DialogUnit** as well as in **ErrorUnit** and **MacExtras**. Be sure to have all three units on volume Programs in folder Units.F, or change lines 1, 2, 17–19 in Listing 6.17 to use different names.

To run the example, type in Listing 6.16, save as **DATAENTRY.R**, and com-

Name: <input type="text" value="Frederholtz, Gypsy Venus"/>			002
Address: <input type="text" value="254 South Broad Street"/>			
City: <input type="text" value="Transylvania"/>	ST: <input type="text" value="PA"/>	ZIP: <input type="text" value="12345"/>	
Telephone: <input type="text" value="(800) 555-1212"/>	Payments: <input type="text" value="457.95"/>		
Birth Date: <input type="text" value="14-Feb-53"/>	Charges: <input type="text" value="550.00"/>		
<input type="button" value="Cancel"/>	<input type="button" value="Previous"/>	<input type="button" value="Next"/>	

Figure 6.15 Modeless dialogs make good-looking data entry forms as in this sample display from Listing 6.17, **DataEntry**.

pile with RMaker, creating the program's resources. Then type in Listing 6.17 and save as DATAENTRY.PAS. Compile with Turbo to run the program, which creates a data file Names.DATA on the current volume.

To type in names and addresses, use the Tab key to move from field to field or click the mouse pointer in the field you want to edit next. If you have a Macintosh Plus or a numeric keypad, you can use the arrow keys to move the vertical bar cursor from one character to another within the current field. All standard editing commands are active. To cut and paste between fields or between records, select any field by clicking and dragging the cursor over it, type Command-C to copy (or choose the Edit menu's Copy command), and Command-V (or Paste) to duplicate that field in subsequent records. Undo throws away changes as long as you do not leave the current field.

Pressing Return or Enter saves the record on screen and moves to the next one. Clicking the Next button in the lower left corner is identical to pressing either of those two keys. Clicking Previous also saves the current record but moves to the preceding one. Clicking Cancel throws out all changes made to all fields in the current record, redisplaying whatever was there before.

The number in the top right corner corresponds with the record on display (002 in the figure). You cannot type over this number—it represents the position of the record in the data file. In this version, you can have from 1 to 100 records. If you need more, change the constant at line 65 in Listing 6.17. (Do this *before* running the program the first time.) Because of the limitation that no data structure can exceed 32,767 bytes, you can have at most about 250, 130-byte records before you exceed Turbo's limit. To go beyond that limit, you would have to read and write individual records on disk rather than read them all into memory as in this example.

Listing 6.16. DATAENTRY.R

```

1: *-----*
2: * DataEntry.PAS resources -- Compile with RMaker *
3: *-----*
4:
5: Programs:Dialogs.F:DataEntry.RSRC      ;; Send output to here
6:
7:
8: *-----*
9: * About box string list *
10: *-----*
11:
12: TYPE STR#
13:     ,1 (32)
14: 6
15: Data Entry Example
16: by Tom Swan
17: Version 1.00
18: (C) 1987 by Swan Software
19: P. O. Box 206, Lititz, PA 17543
20: (717)-627-1911
21:
22:

```

```

23: *-----*
24: * The Apple Info menu *
25: *-----*
26:
27: TYPE MENU
28: ,1
29: \14
30: About DataEntry...
31: (-
32:
33:
34: *-----*
35: * The File menu *
36: *-----*
37:
38: TYPE MENU
39: ,2
40: File
41: Close
42: Save /S
43: (-
44: Quit /Q
45:
46:
47: *-----*
48: * The Edit menu *
49: *-----*
50:
51: TYPE MENU
52: ,3
53: Edit
54: Undo /Z
55: (-
56: Cut /X
57: Copy /C
58: Paste /V
59: Clear
60:
61:
62: TYPE MENU
63: ,4
64: Record
65: Next /N
66: Previous /P
67: (-
68: Number...
69:
70:
71: *-----*
72: * The entry-form dialog *
73: *-----*
74:
75: type DLOG
76: ,1000
77: Entry form
78: 40 18 316 494
79: Visible NoGoAway
80: 1
81: 0
82: 1000
83:
84:
85: type DITL ;; Item list for entry form dialog
86: ,1000 (32)
87: 22 ;; Number of items following
88:

```

(continued)

298 Programming with Macintosh Turbo Pascal

```
89: BtnItem Enabled      ;; 1. Next button
90: 250 348 266 448
91: Next
92:
93: BtnItem Enabled      ;; 2. Cancel button
94: 250 21 266 121
95: Cancel
96:
97: BtnItem Enabled      ;; 3. Previous button
98: 250 215 266 315
99: Previous
100:
101: EditText Enabled    ;; 4-12. Field entry areas
102: 20 105 36 445
103:
104:
105: EditText Enabled
106: 55 105 71 445
107:
108:
109: EditText Enabled
110: 90 105 106 205
111:
112:
113: EditText Enabled
114: 90 260 106 310
115:
116:
117: EditText Enabled
118: 90 360 106 445
119:
120:
121: EditText Enabled
122: 145 105 161 240
123:
124:
125: EditText Enabled
126: 180 105 196 240
127:
128:
129: EditText Enabled
130: 145 350 161 445
131:
132:
133: Ed1 Text Enabled
134: 180 50 196 445
135:
136:
137: StatText Disabled    ;; 13. Record number in upper right corner
138: -3 452 12 480
139: 001
140:
141: StatText Disabled    ;; 14-22 Field names for each edit area
142: 20 25 36 85
143: Name:
144:
145: StatText Disabled
146: 55 25 71 90
147: Address:
148:
149: StatText Disabled
150: 90 25 106 90
151: City:
152:
153: StatText Disabled
154: 90 235 106 255
155: ST:
156:
```

```

157: StatText Disabled
158: 90 325 106 355
159: ZIP:
160:
161: StatText Disabled
162: 145 25 161 100
163: Telephone:
164:
165: StatText Disabled
166: 180 25 196 100
167: Birth Date:
168:
169: StatText Disabled
170: 145 270 161 345
171: Payments:
172:
173: StatText Disabled
174: 180 270 196 345
175: Charges:
176:
177:
178: *-----*
179: * The Save Changes? alert *
180: *-----*
181:
182: type ALRT
183:      ,1001 (4)
184: 73 105 198 400
185: 1001
186: 5555
187:
188:
189: type DITL      ;; Item list for alert
190:      ,1001 (32)
191: 4      ;; Number of items following
192:
193: BtnItem Enabled      ;; 1. Yes button
194: 60 25 82 105
195: Yes
196:
197: BtnItem Enabled      ;; 2. Cancel button
198: 95 200 117 280
199: Cancel
200:
201: BtnItem Enabled      ;; 3. No button
202: 95 25 117 105
203: No
204:
205: StatText Disabled      ;; 4. Message text
206: 7 86 27 295
207: Save changes before quitting?
208:
209:
210:
211: *-----*
212: * The Enter record number dialog *
213: *-----*
214:
215: type DLOG
216:      ,1002
217: RecNum
218: 100 138 220 376
219: Visible NoGoAway
220: 1
221: 0
222: 1002
223:
224:

```

(continued)

```

225: type DITL
226:      ,1002 (32)
227: 4          ;; Number of items following
228:
229: BtnItem Enabled      ;; 1. Ok button
230: 86 25 104 99
231: Ok
232:
233: BtnItem Enabled      ;; 2. Cancel button
234: 86 141 104 215
235: Cancel
236:
237: StatText Disabled    ;; 3. Prompt
238: 17 25 35 146
239: Record number?
240:
241: EditText Enabled     ;; 4. Entry area. (Default number = 1).
242: 50 105 66 131
243: 1
244:
245:
246: *-----*
247: * The error alert template                                *
248: *-----*
249:
250: TYPE ALRT
251:      ,999 (4)          ;; Resource ID, (4) = preload
252: 45 28 154 481          ;; Top Left Bottom Right
253: 999                    ;; Item list ID (following)
254: 5555                   ;; Stages (none)
255:
256:
257: TYPE DITL              ;; Error alert item list
258:      ,999 (32)          ;; Resource ID , (30)=purgeable
259: 4
260:
261: BtnItem Enabled      ;; 1. Ok button
262: 71 360 103 440
263: Ok
264:
265: StatText Disabled    ;; 2. Error number
266: 25 80 41 165
267: Error ^0:
268:
269: StatText Disabled    ;; 3. Error message
270: 25 170 41 440
271: ^1
272:
273: StatText Disabled    ;; 4. Help message
274: 80 15 96 350
275: ^2
276:
277:
278: * END

```

Listing 6.17. DATAENTRY.PAS

```

1: {$O Programs:Dialogs.F: }           { Send compiled code to here }
2: {$R Programs:Dialogs.F:DataEntry.Rsrc} { Use this compiled resource file }
3: {$U-}                               { Turn off standard library units }
4:
5:
6: PROGRAM DataEntry;
7:
8: (*
9:
10:  * PURPOSE : Data entry example using modeless dialog
11:  * SYSTEM  : Macintosh / Turbo Pascal
12:  * AUTHOR   : Tom Swan
13:
14:  *)
15:
16:
17: {$U Programs:Units.F:MacExtras }      { Open these library unit files }
18: {$U Programs:Units.F:DialogUnit }
19: {$U Programs:Units.F:ErrorUnit }
20:
21:
22:     USES
23:
24:         PasInOut,
25:         MemTypes, QuickDraw, OSIntf, ToolIntf, PackIntf, MacExtras,
26:         DialogUnit, ErrorUnit;
27:
28:
29:
30:     CONST
31:
32:
33:         FileID      = 2;      { Resource ID number for File menu }
34:         CloseCmd     = 1;
35:         SaveCmd      = 2;
36:         {-----}
37:         QuitCmd      = 4;
38:
39:         RecordID     = 4;
40:         NextCmd       = 1;
41:         PrevCmd       = 2;
42:         {-----}
43:         NumbCmd       = 4;
44:
45:
46:     { Dialog and Alert resource IDs and item numbers }
47:
48:         EntryID       = 1000; { Resource ID of Entry dialog }
49:         SaveID         = 1001; { Resource ID of Save Changes? alert }
50:         RecNumID       = 1002; { Resource ID of Record number? dialog }
51:
52:         NextButton     = Ok;    { Dialog "Next" button number (same as Ok) }
53:         PrevButton     = 3;     { Dialog "Previous" button number }
54:         FirstEditItem  = 4;     { Number of first edit text item }
55:
56:
57:     { Keyboard scan codes -- not the same as ASCII values }
58:
59:         KeyReturn      = 36;    { Return key code }
60:         KeyEnter       = 76;    { Enter key code }
61:
62:
63:     { Data file constants }
64:

```

(continued)

```

65:      MaxRec      = 100;          { Maximum records in database }
66:      FileName    = 'Names.DATA'; { Name of disk file }
67:
68:      NameLen      = 32;          { DataRec string lengths }
69:      AddressLen   = 32;
70:      CityLen      = 15;
71:      STLen        = 5;
72:      ZipLen       = 5;
73:      PhoneLen     = 14;
74:      DateLen      = 8;
75:      NumLen       = 11;
76:
77:      NameItem     = 1;          { Dialog edit text numbers for each field }
78:      AddressItem  = 2;          { in the DataRec. }
79:      CityItem     = 3;
80:      STItem       = 4;
81:      ZipItem      = 5;
82:      PhoneItem    = 6;
83:      BirthdateItem = 7;
84:      PaymentsItem = 8;
85:      ChargesItem  = 9;
86:
87:      MaxField     = 9;          { The number of fields specified above }
88:
89:
90:
91:  TYPE
92:
93:      Dollar = LONGINT; { Amounts in approx. range of +/- $21,474,836.46 }
94:
95:      DataRec =
96:          RECORD
97:              Name      : String[ NameLen ];
98:              Address   : String[ AddressLen ];
99:              City      : String[ CityLen ];
100:             ST        : String[ STLen ];
101:             Zip       : String[ ZipLen ];
102:             Phone     : String[ PhoneLen ];
103:             BirthDate : String[ DateLen ];
104:             Payments  : Dollar;
105:             Charges   : Dollar;
106:         END; { DataRec }
107:
108:      DataArray = ARRAY[ 1 .. MaxRec ] OF DataRec;
109:      DataArrayPtr = ^DataArray;
110:
111:
112:  VAR
113:
114:      recordMenu    : MenuHandle;      { Pull-down menu handle }
115:
116:      iBeam         : CursHandle;      { I-beam text entry cursor }
117:      watch         : CursHandle;      { Watch (busy) cursor }
118:      cross         : CursHandle;      { Cross (button selector) cursor }
119:      theCursor     : CursHandle;      { The current cursor. NIL=arrow. }
120:
121:      editArea      : Rect;            { Screen areas where cursor }
122:      buttonArea    : Rect;            { shape should change. }
123:
124:      dPtr          : DialogPtr;       { Entry form dialog pointer }
125:
126:      quitRequested : BOOLEAN;          { TRUE if quitting }
127:      dataDirty     : BOOLEAN;          { TRUE if changes not saved }
128:      theData       : DataArrayPtr;    { Pointer to data array }
129:      dataIndex     : INTEGER;          { Index into theData array }
130:
131:

```

```

132: { The following array holds handles to each edit text item in
133: the dialog entry form. Copying the handles into a global array
134: helps cut down the number of calls to the Dialog manager. }
135:
136: ItemHandles :
137:     ARRAY[ 1 .. MaxField ] OF Handle;
138:
139:
140:
141: PROCEDURE ResetCursor;
142:
143: { Change cursor into its standard shape (an arrow) }
144:
145: BEGIN
146:     InitCursor;
147:     theCursor := NIL      { Means "arrow cursor" to other procedures }
148: END; { ResetCursor }
149:
150:
151: PROCEDURE InitNewArray;
152:
153: { Create new array of blank records }
154:
155: BEGIN
156:
157: { Use the following statement to quickly clear large arrays. But
158: be careful--it just fills the entire array with zero bytes with
159: no regard for the various field types in the records. }
160:
161:     FillChar( theData^, sizeof( DataArray ), 0 )
162:
163: END; { InitNewArray }
164:
165:
166: PROCEDURE ShowRecordNumber;
167:
168: { Display current record number in dialog window }
169:
170: VAR
171:
172:     itemNo    : INTEGER;
173:     itemType  : INTEGER;
174:     item      : Handle;
175:     itemRect  : Rect;
176:     s         : Str255;
177:
178: BEGIN
179:
180: { Calculate record number position in dialog item list }
181:
182:     itemNo := FirstEditItem + MaxField;
183:
184:     GetDItem( dPtr, itemNo, itemType, item, itemRect );
185:     NumToString( dataIndex, s ); { Convert number to string }
186:     WHILE length( s ) < 3 DO { Expand to three digits: }
187:         insert( '0', s, 1 ); { 001, 002, ..., 025, etc. }
188:     SetIText( item, s ) { Store string s into dialog item }
189:
190: END; { ShowRecordNumber }
191:
192:
193: PROCEDURE DollarToStr( d : Dollar; VAR s : Str255 );
194:
195: { Convert Dollar amount d to a string s }
196:

```

(continued)


```

197:   VAR
198:
199:       negative : BOOLEAN;
200:
201:   BEGIN
202:       negative := ( d < 0 );           { Remember if value is negative }
203:       d := ABS( d );                 { Convert value to positive }
204:       NumToString( d, s );           { Do raw conversion to string }
205:       WHILE length( s ) < 3 DO
206:           Insert( '0', s, 1 );       { Make at least 3 digits long }
207:       Insert( '.', s, length( s ) - 1 ); { Insert decimal point }
208:       IF negative
209:           THEN Insert( '-', s, 1 )   { Insert minus sign }
210:       END; { DollarToString }
211:
212:
213: PROCEDURE StrToDollar( s : Str255; VAR d : Dollar );
214:
215: { Convert string s to dollar value d }
216:
217:   VAR
218:
219:       p : INTEGER;   { Position of decimal point in string }
220:
221:
222:   PROCEDURE FindDecimal;
223:
224: { Set global variable p to position of decimal point in string s
225:   or set it to zero if there is no decimal }
226:
227:   BEGIN
228:       p := pos( '.', s )
229:   END; { FindDecimal }
230:
231:   BEGIN
232:       FindDecimal;           { Remove any decimal point }
233:       WHILE p > 0 DO
234:           BEGIN
235:               Delete( s, p, 1 );
236:               FindDecimal
237:           END; { while }
238:       StringToNum( s, d )    { Convert to Dollar type }
239:   END; { StrToDollar }
240:
241:
242: PROCEDURE FieldToDialog( field : Str255; itemNum : INTEGER );
243:
244: { Save this string in the dialog edit text with this item number }
245:
246:   BEGIN
247:       SetIText( itemHandles[ itemNum ], field )
248:   END; { FieldToDialog }
249:
250:
251: PROCEDURE DialogToField( VAR field : Str255; len, itemNum : INTEGER );
252:
253: { Retrieve dialog itemNum EditText and return up to len chars in field }
254:
255:   BEGIN
256:       GetIText( itemHandles[ itemNum ], field );
257:       IF Length( field ) > len
258:           THEN field := copy( field, 1, len )
259:   END; { DialogToField }
260:
261:

```

```

262: PROCEDURE RecToDialog;
263:
264: { Disassemble current record into dialog text items for editing }
265:
266:   VAR
267:
268:     s : Str255;
269:
270:   BEGIN
271:     SelIText( dPtr, FirstEditItem, 0, 0 );      { Un hilite first field }
272:     WITH theData^[ dataIndex ] DO
273:       BEGIN
274:         FieldToDialog( name, NameItem );          { Copy fields to dialog }
275:         FieldToDialog( address, AddressItem );    { edit text items. }
276:         FieldToDialog( city, CityItem );
277:         FieldToDialog( ST, StItem );
278:         FieldToDialog( Zip, ZipItem );
279:         FieldToDialog( Phone, PhoneItem );
280:         FieldToDialog( BirthDate, BirthdateItem );
281:         DollarToStr( Payments, s );               { Convert fields not }
282:         FieldToDialog( s, PaymentsItem );         { already strings }
283:         DollarToStr( Charges, s );
284:         FieldToDialog( s, ChargesItem )
285:       END; { with }
286:       SelIText( dPtr, FirstEditItem, 0, MaxInt ); { Hilite first field }
287:       ShowRecordNumber
288:     END; { RecToDialog }
289:
290:
291: PROCEDURE DialogToRec;
292:
293: { Assemble dialog edit text items into the current Pascal record }
294:
295:   VAR
296:
297:     s : Str255;
298:
299:   BEGIN
300:     WITH theData^[ dataIndex ] DO
301:       BEGIN
302:         DialogToField( s, NameLen, NameItem );    { Get edited text item }
303:         name := s;                                { Copy to record field }
304:         DialogToField( s, AddressLen, AddressItem );
305:         address := s;
306:         DialogToField( s, CityLen, CityItem );
307:         city := s;
308:         DialogToField( s, StLen, StItem );
309:         ST := s;
310:         DialogToField( s, ZipLen, ZipItem );
311:         Zip := s;
312:         DialogToField( s, PhoneLen, PhoneItem );
313:         Phone := s;
314:         DialogToField( s, DateLen, BirthdateItem );
315:         BirthDate := s;
316:         DialogToField( s, NumLen, PaymentsItem );
317:         StrToDollar( s, Payments );               { Convert dollar str }
318:         DialogToField( s, NumLen, ChargesItem );  { to Dollar type }
319:         StrToDollar( s, Charges );
320:       END { with }
321:     END; { DialogToRec }
322:
323:
324: PROCEDURE NewRecord( recNum : LONGINT );
325:
326: { Save current record and display another }
327:

```

(continued)

```

328: BEGIN
329:     DialogToRec;                { Save current record }
330:     IF recNum < 1                { Limit record number range }
331:     THEN                        { to 1 .. MaxRec. }
332:         recNum := MaxRec
333:     ELSE
334:         IF recNum > MaxRec
335:         THEN
336:             recNum := 1;
337:             dataIndex := RecNum;    { Change record number }
338:             RecToDialog            { Disassemble record into edit fields }
339:         END; { NewRecord }
340:
341:
342: PROCEDURE DoNext;
343:
344: { Advance to next record, saving contents of the current record. }
345:
346: BEGIN
347:     NewRecord( dataIndex + 1 )
348: END; { DoNext }
349:
350:
351: PROCEDURE DoPrevious;
352:
353: { Go back to previous record, saving contents of the current record }
354:
355: BEGIN
356:     NewRecord( dataIndex - 1 )
357: END; { DoPrevious }
358:
359:
360: PROCEDURE DoSave;
361:
362: { Save records in a disk file }
363:
364: VAR
365:
366:     dataFile      : FILE OF DataRec;
367:     errCode       : INTEGER;
368:     rn            : INTEGER;
369:
370: BEGIN
371:
372:     { Before writing records to disk, insert the current fields now on
373:     display into theData array. If you didn't do this, changes to the
374:     displayed record would be lost because they normally are saved only
375:     after you press return (or click the previous or next buttons). }
376:
377:     DialogToRec;
378:
379:
380:     SetCursor( watch^^ );    { Display wrist watch while writing }
381:
382:     {$i-} Rewrite( dataFile, FileName ); {$i+}
383:     errCode := IoResult;
384:     IF errCode = 0 THEN
385:     BEGIN
386:         rn := 1;
387:         WHILE ( errCode = 0 ) AND ( rn <= MaxRec ) DO
388:         BEGIN
389:             {$i-} write( dataFile, theData^[ rn ] ); {$i+}
390:             errCode := IoResult;
391:             rn := rn + 1
392:         END { while }
393:     END; { if }
394:

```

```

395:     ResetCursor;
396:
397:     IF errCode <> 0
398:         THEN IoError( errCode, 'Check disk and try again' )
399:         ELSE dataDirty := FALSE;
400:
401:     { $i- } Close( dataFile ); { $i+ }
402:     errCode := IoResult
403:
404: END; { DoSave }
405:
406:
407: PROCEDURE ReadFromDisk;
408:
409: { Read disk file into data array -- file already known to exist }
410:
411:     VAR
412:
413:         dataFile      : FILE OF DataRec;
414:         errCode        : INTEGER;
415:         rn             : INTEGER;
416:
417:     BEGIN
418:
419:         { $i- } Reset( dataFile, FileName ); { $i+ }
420:         errCode := IoResult;
421:         IF errCode <> 0 THEN InitNewArray ELSE      { If no file, start one }
422:         BEGIN
423:             SetCursor( watch^^ );    { Display wrist watch while reading }
424:             rn := 1;
425:             WHILE ( errCode = 0 ) AND ( rn <= MaxRec ) DO
426:             BEGIN
427:                 { $i- } read( dataFile, theData^[ rn ] ); { $i+ }
428:                 errCode := IoResult;
429:                 rn := rn + 1
430:             END; { while }
431:             ResetCursor;                { Make cursor an arrow again }
432:             IF errCode <> 0
433:                 THEN IoError( errCode, '' );
434:             Close( dataFile )
435:         END; { if }
436:         dataIndex := 1;      { Start new display with first record }
437:         dataDirty := FALSE;  { Mark data "saved" }
438:         RecToDialog          { Display record for editing }
439:
440:     END; { ReadFromDisk }
441:
442:
443: PROCEDURE DoNumber;
444:
445: { Request record number and advance to that record saving
446:   current record contents }
447:
448:     CONST
449:
450:         NewNumItem = 4;    { Edit text item number }
451:
452:     VAR
453:
454:         ndPtr      : DialogPtr;      { Number-Dialog Pointer }
455:         itemType    : INTEGER;
456:         itemHandle  : Handle;
457:         itemRect    : Rect;
458:         itemHit     : INTEGER;
459:         newNum      : LONGINT;
460:         s            : Str255;
461:

```

(continued)

```

462: BEGIN
463:
464:     ndPtr := GetNewDialog( RecNumID, NIL, POINTER(-1) );
465:     IF ndPtr <> NIL THEN
466:         BEGIN
467:
468:             OutlineOk( ndPtr );
469:             GetDItem( ndPtr, NewNumItem, itemType, itemHandle, itemRect );
470:             SelItem( ndPtr, NewNumItem, 0, MaxInt );
471:             REPEAT
472:                 ModalDialog( NIL, itemHit )
473:             UNTIL ( itemHit = Ok ) OR ( itemHit = Cancel );
474:             IF itemHit = Ok THEN
475:                 BEGIN
476:                     GetIText( itemHandle, s );      { Get edited text }
477:                     StringToNum( s, newNum );      { Convert to number }
478:                     NewRecord( newNum )            { Display the new record }
479:                 END; { if }
480:
481:                 DisposDialog( ndPtr )             { Remove dialog from memory }
482:
483:             END { if }
484:
485:         END; { DoNumber }
486:
487:
488: PROCEDURE DoClose;
489:
490: { Respond to File menu Close command }
491:
492: BEGIN
493:     IF FrontWindow <> dPtr
494:     THEN CloseDAWindow           { Close desk accessory window }
495:     END; { DoClose }
496:
497:
498: PROCEDURE DoFileMenuCommands( cmdNumber : INTEGER );
499:
500: { Execute command in the File menu }
501:
502: BEGIN
503:     CASE cmdNumber OF
504:         CloseCmd : DoClose;
505:         SaveCmd  : IF dataDirty THEN DoSave;
506:         QuitCmd  : quitRequested := TRUE
507:     END { case }
508:     END; { DoFileMenuCommands }
509:
510:
511: PROCEDURE DoEditMenuCommands( cmdNumber : INTEGER );
512:
513: { Execute command in the Edit menu }
514:
515: BEGIN
516:     IF NOT SystemEdit( cmdNumber - 1 ) THEN
517:         BEGIN
518:
519:             dataDirty := TRUE;           { Assume editing done. }
520:
521:             CASE cmdNumber OF
522:                 UndoCmd    : SysBeep( 4 );      { Undo doesn't. }
523:                 CutCmd     : DlgCut( dPtr );     { These procedures affect }
524:                 CopyCmd    : DlgCopy( dPtr );    { selected text }
525:                 PasteCmd   : DlgPaste( dPtr );   { in the dialog edit }
526:                 ClearCmd   : DlgDelete( dPtr )   { text fields. }
527:             END { case }
528:
529:         END { if }
530:     END; { DoEditMenuCommands }

```

```

531:
532:
533: PROCEDURE DoRecordMenuCommands( cmdNumber : INTEGER );
534:
535: { Execute Record menu command }
536:
537: BEGIN
538:     CASE cmdNumber OF
539:         NextCmd    : DoNext;
540:         PrevCmd    : DoPrevious;
541:         NumbCmd    : DoNumber
542:     END { case }
543: END; { DoRecordMenuCommands }
544:
545:
546: PROCEDURE DoCommand( command : LongInt );
547:
548: { Execute command returned by MenuSelect or MenuKey functions }
549:
550: VAR
551:
552:     whichMenu    : INTEGER;    { Menu number of selected command }
553:     whichItem    : INTEGER;    { Menu item number of command }
554:
555: BEGIN
556:
557:     whichMenu := HiWord( command );
558:     whichItem := LoWord( command );
559:
560:     CASE whichMenu OF
561:         AppleID    : DoAppleMenuCommands( whichItem );
562:         FileID     : DoFileMenuCommands( whichItem );
563:         EditID     : DoEditMenuCommands( whichItem );
564:         RecordID   : DoRecordMenuCommands( whichItem );
565:     END; { case }
566:
567:     HiliteMenu( 0 ) { Unhighlight menu title }
568:
569: END; { DoCommand }
570:
571:
572: PROCEDURE MouseDownEvents;
573:
574: { Someone pressed the mouse button. Check its location and respond. }
575:
576: VAR
577:
578:     partCode : INTEGER;    { Identifies what item was clicked. }
579:
580: BEGIN
581:
582:     WITH theEvent DO
583:
584:         BEGIN
585:
586:             partCode := FindWindow( where, whichWindow );
587:
588:             CASE partCode OF
589:
590:                 inMenuBar
591:                     : DoCommand( MenuSelect( where ) );
592:
593:                 inSysWindow
594:                     : SystemClick( theEvent, whichWindow );
595:
596:             END { case }
597:
598:         END; { with }
599:
600: END; { MouseDownEvents }

```

(continued)

```

601:
602:
603: PROCEDURE KeyDownEvents;
604:
605: { A key was pressed. Do something with incoming character. }
606:
607:   VAR
608:
609:     ch : CHAR;
610:
611:   BEGIN
612:     WITH theEvent DO
613:       BEGIN
614:         ch := CHR( BitAnd( message, charCodeMask ) );
615:
616:         IF BitAnd( modifiers, CmdKey ) <> 0
617:           THEN DoCommand( MenuKey( ch ) )      { Execute a command }
618:
619:         END { with }
620:       END; { KeyDownEvents }
621:
622:
623: PROCEDURE DialogEvents;
624:
625: { Handle events recognized as belonging to the data
626:   entry modeless dialog. }
627:
628:   VAR
629:
630:     itemHit : INTEGER;
631:     flag    : BOOLEAN;
632:
633:
634:   PROCEDURE TranslateChar( message : LONGINT );
635:
636:   { Translate character reported by a keydown event into an
637:     equivalent itemHit value. This lets people use the keyboard or
638:     the mouse to click buttons in the entry form dialog. Chars that
639:     are not translated are passed to DialogSelect later. }
640:
641:   VAR
642:
643:     keyCode : LONGINT;      { Not the same as an ASCII value! }
644:
645:   BEGIN
646:     keyCode := BitAnd( message, keyCodeMask ) DIV 256;
647:     IF ( keyCode = KeyReturn ) OR ( keyCode = KeyEnter )
648:       THEN itemHit := Ok    { Simulate click on the Ok button }
649:     END; { TranslateChar }
650:
651:
652:   BEGIN
653:
654:     itemHit := Ok - 1;      { Any value <> Ok is Okay }
655:
656:
657:   { If event is a keypress, check whether command key is down. If it
658:     is, then ignore it -- command keys should not be passed to the
659:     Dialog manager. They will be handled in the usual way. }
660:
661:   WITH theEvent DO
662:     IF what = KeyDown THEN
663:       IF BitAnd( modifiers, CmdKey ) = 0
664:         THEN TranslateChar( message )  { Set itemHit }
665:       ELSE exit; { Ignore menu command keys }
666:
667:

```

```

668:  { Check via DialogSelect whether an enabled item was clicked, enter
669:    chars into fields, etc. But if the "next" button was clicked by
670:    pressing return or enter, don't call DialogSelect. This avoids
671:    passing the characters we want to process here--cr and enter--rather
672:    than passing them to the dialog manager. }
673:
674:    IF itemHit = NextButton
675:      THEN flag := TRUE
676:      ELSE flag := DialogSelect( theEvent, dPtr, itemHit );
677:
678:
679:  { Finish processing Dialog event by checking for clicks on the Ok or
680:    Cancel buttons, and take appropriate actions. }
681:
682:    IF flag THEN
683:
684:      CASE itemHit OF
685:
686:        NextButton   : DoNext;
687:
688:        PrevButton   : DoPrevious;
689:
690:        Cancel       : RecToDialog
691:
692:        OTHERWISE dataDirty := TRUE      { Assume a change was made }
693:
694:      END { case }
695:
696:    END; { DialogEvents }
697:
698:
699:  PROCEDURE SetUpCursors;
700:
701:  { Initialize cursor patterns }
702:
703:  BEGIN
704:    iBeam := GetCursor( IBeamCursor );
705:    watch := GetCursor( WatchCursor );
706:    cross := GetCursor( CrossCursor );
707:    ResetCursor
708:  END; { SetUpCursors }
709:
710:
711:  PROCEDURE SetUpMenuBar;
712:
713:  { Initialize and display menu bar }
714:
715:  BEGIN
716:
717:    appleMenu := GetMenu( AppleID ); { Read menu resources }
718:    fileMenu  := GetMenu( FileID );
719:    editMenu  := GetMenu( EditID );
720:    recordMenu := GetMenu( RecordID );
721:
722:    InsertMenu( appleMenu, 0 );      { Insert into menu list }
723:    InsertMenu( fileMenu, 0 );
724:    InsertMenu( editMenu, 0 );
725:    InsertMenu( recordMenu, 0 );
726:
727:    AddResMenu( appleMenu, 'DRVr' ); { Add desk accessory names }
728:
729:    DrawMenuBar      { Display the menu bar }
730:
731:  END; { SetUpMenuBar }
732:
733:

```

(continued)


```

734: PROCEDURE SetUpDialog;
735:
736: { Initialize the entry form dialog and cursor shape areas }
737:
738:   VAR
739:
740:     itemType : INTEGER;
741:     item      : Handle;
742:     itemRect  : Rect;
743:     itemNo    : INTEGER;
744:
745:   BEGIN
746:
747:
748:     dPtr := GetNewDialog( EntryID, NIL, POINTER(-1) );
749:     IF dPtr = NIL
750:       THEN ExitToShell;    { End program -- dialog not available }
751:
752:
753:     { Copy handles of each edit text item in the dialog. This makes
754:       transferring strings to and from the dialog easier and avoids
755:       calling GetDItem too often. }
756:
757:     FOR itemNo := 0 TO MaxField - 1 DO
758:       BEGIN
759:         GetDItem( dPtr, itemNo + FirstEditItem,
760:                   itemType, item, itemRect );
761:         IF item = NIL
762:           THEN ExitToShell;    { A field is missing! }
763:         itemHandles[ itemNo + 1 ] := item
764:       END; { for }
765:
766:
767:     { Initialize area rectangles in which the cursor takes different
768:       shapes to indicate what you can do. Assumes that all buttons are
769:       the lowest objects below the same horizontal plane. This plan may
770:       not work if you change the location of objects in the dialog. }
771:
772:     editArea := dPtr^.portRect;
773:     buttonArea := editArea;
774:
775:     GetDItem( dPtr, Ok, itemType, item, itemRect ); { Ok button }
776:
777:     WITH itemRect DO
778:       BEGIN
779:         top := top - 4;
780:         editArea.bottom := top;
781:         buttonArea.top := top
782:       END { with }
783:
784:   END; { SetUpDialog }
785:
786:
787: PROCEDURE SetUpArray;
788:
789: { Initialize data array as non-relocatable object on heap }
790:
791:   BEGIN
792:     theData := DataArrayPtr( NewPtr( SizeOf( DataArray ) ) );
793:     IF MemError <> NoErr THEN
794:       BEGIN
795:         DisplayError( 1, 'Not enough memory',
796:                       'Set MaxRec to lower value', StopError );
797:         ExitToShell
798:       END; { if }
799:   END; { SetUpArray }
800:
801:

```

```

802: PROCEDURE Initialize;
803:
804: { Program calls this routine one time at start }
805:
806: BEGIN
807:     SetUpCursors;           { Initialize various cursor patterns }
808:     SetUpMenuBar;          { Initialize and display menus }
809:     DisplayAboutBox;       { Identify program }
810:     SetUpDialog;           { Initialize data entry dialog }
811:     SetUpArray;            { Initialize data array }
812:     ReadFromDisk;          { Read records or start new file }
813:     quitRequested := FALSE { TRUE on selecting Quit command }
814: END; { Initialize }
815:
816:
817: FUNCTION QuitConfirmed : BOOLEAN;
818:
819: { The program's "deinitialization" routine. }
820: { Returns TRUE if it's okay to quit program }
821:
822: CONST
823:
824:     Yes = Ok;    { Button numbers }
825:     No  = 3;
826:
827: VAR
828:
829:     itemHit : INTEGER;
830:
831: BEGIN
832:     IF QuitRequested THEN { Quit command chosen }
833:     BEGIN
834:         ResetCursor;      { Change cursor to standard arrow }
835:         IF dataDirty THEN { Need confirmation if editing was done }
836:         BEGIN
837:             itemHit := CautionAlert( SaveID, NIL ); { Display alert }
838:             IF itemHit = Cancel
839:             THEN
840:                 quitRequested := FALSE { Quit not confirmed }
841:             ELSE
842:                 IF ItemHit = Yes
843:                 THEN
844:                     BEGIN
845:                         DoSave; { Try to save records }
846:                         quitRequested := NOT dataDirty { TRUE if that worked }
847:                     END
848:                 END { if }
849:             END; { if }
850:             QuitConfirmed := quitRequested { Pass result as function value }
851:         END; { QuitConfirmed }
852:
853:
854: PROCEDURE ChangeCursor;
855:
856: { Change cursor shape according to its position }
857:
858: VAR
859:
860:     NewCursor : CursHandle;
861:     mouseLocation : Point;
862:     oldPort : GrafPtr;
863:
864: BEGIN
865:
866: { If another window is frontmost, exit this procedure. This lets
867:   desk accessories handle their own cursor changes. }
868:

```

(continued)

```

869:      IF FrontWindow <> dPtr THEN Exit;
870:
871:
872:      { All points must be in reference to the dialog window. First save the
873:      old port for later restoring, then set the port to the dialog record
874:      pointer. }
875:
876:      GetPort( oldPort ); SetPort( dPtr );
877:
878:
879:      { Locate the mouse in local coordinates, that is, relative to the
880:      current grafPort--the dialog window. }
881:
882:      GetMouse( mouseLocation );
883:
884:
885:      { Check whether mouse is inside one of the areas in which its shape
886:      changes to indicate what you can do there. }
887:
888:      IF PtInRect( mouseLocation, editArea )      { I Beam for edit area }
889:      THEN NewCursor := iBeam ELSE
890:
891:      IF PtInRect( mouseLocation, buttonArea ) { Cross for the buttons }
892:      THEN NewCursor := cross
893:      ELSE NewCursor := NIL;                    { Arrow for other areas }
894:
895:
896:      { If the above logic calls for a change in the cursor shape, then
897:      set the cursor to a new shape. Otherwise, do nothing. Doing this
898:      avoids repeated calls to SetCursor every time through this loop. }
899:
900:
901:      IF NewCursor <> theCursor THEN
902:      BEGIN
903:          IF NewCursor = NIL
904:          THEN InitCursor                      { Change to standard arrow }
905:          ELSE SetCursor( NewCursor^^ );      { Change to another shape }
906:          theCursor := NewCursor              { Remember current shape }
907:      END; { if }
908:
909:
910:      { Restore the original port, saved on entry to this procedure. }
911:
912:      SetPort( oldPort );
913:
914:      END; { ChangeCursor }
915:
916:
917: FUNCTION IsProgramEvent(      eventMask : INTEGER;
918:                          VAR theEvent  : EventRecord ) : BOOLEAN;
919:
920: { Returns true if GetNextEvent returns true. Also handles dialog events. }
921:
922: BEGIN
923:
924:      { Note: function IsDialogEvent must be called after GetNextEvent even
925:      if that function returns false. Otherwise, null events will not be
926:      send to IsDialogEvent and the cursor will not blink. }
927:
928:      IsProgramEvent := GetNextEvent( eventMask, theEvent );
929:      IF IsDialogEvent( theEvent )
930:      THEN DialogEvents
931:
932:      END; { IsProgramEvent }
933:

```

```

934:
935: PROCEDURE DoSystemTasks;
936:
937: { Do operations at each pass through main program loop }
938:
939: BEGIN
940:
941:     ChangeCursor; { Reshape cursor according to its position }
942:
943:     SystemTask;   { Give DAs their fair share of time }
944:
945:     IF FrontWindow = dPtr THEN
946:
947:         BEGIN { Set up menu commands for data entry window }
948:
949:             DisableItem( fileMenu, CloseCmd );
950:             EnableItem(  fileMenu, SaveCmd );
951:
952:             DisableItem( editMenu, UndoCmd );
953:
954:             EnableItem(  recordMenu, NextCmd );
955:             EnableItem(  recordMenu, PrevCmd );
956:             EnableItem(  recordMenu, NumbCmd );
957:
958:         END ELSE
959:
960:         IF FrontWindow <> dPtr THEN
961:
962:             BEGIN { Set up menu commands for active desk accessory }
963:
964:                 EnableItem( fileMenu, CloseCmd );
965:                 DisableItem( fileMenu, SaveCmd );
966:
967:                 EnableItem( editMenu, UndoCmd );
968:
969:                 DisableItem( recordMenu, NextCmd );
970:                 DisableItem( recordMenu, PrevCmd );
971:                 DisableItem( recordMenu, NumbCmd );
972:
973:             END { else / if }
974:
975:         END; { DoSystemTasks }
976:
977:
978: BEGIN
979:
980:     Initialize;
981:
982:     REPEAT
983:
984:         DoSystemTasks;
985:
986:         IF IsProgramEvent( everyEvent, theEvent ) THEN
987:
988:             CASE theEvent.what OF
989:
990:                MouseDown    : MouseDownEvents;
991:                 KeyDown      : KeyDownEvents;
992:
993:             END { case }
994:
995:         UNTIL QuitConfirmed
996:
997:     END.

```

DataEntry Play-by-Play

DATAENTRY.R (1-278)

You should have little trouble following the resource file listing. It contains no new elements—just more of them. The dialog (71-175) is the data entry form in Figure 6.15, a standard double-border dialog with resource ID 1000. The dialog's item list contains three buttons (89-99), nine **Edit Text** items (101-134), and ten **Stat-Text** items (137-175).

The **Stat Text** and **Edit Text** items form pairs of labels and edit boxes for typing record fields. For example, the **Edit Text** item at lines 101-102 is the typing area for the **Stat Text** field label, Name: at lines 141-143. To design custom forms, replace these items with your own labels and editing areas and change line 87 to the total number of fields in the dialog's item list.

The alert dialog (178-207) displays when you attempt to quit the program after making changes to records. At that time, the program asks whether you want to save your changes, similar to the way Listing 6.6 does. (See also Figure 6.6.)

Another dialog (211-243) appears when you choose the Record menu's Number command. Figure 6.16 shows this dialog in action. It lets you type numbers to locate records at random.

The final resource definition is identical to error alert 999 in Listing 6.14 (48-77). Figure 6.13 illustrates how this alert appears.

DATAENTRY.PAS (1-137)

The program uses nine units (24-26). PasInOut activates Pascal's file package, allowing the program to use **Reset** and **Rewrite** statements to open and create files and **Read** and **Write** to save and recall records on disk.

The constants at lines 33-43 add File and Record menus to the program. Remember that MacExtras (see Chapter 4) defines the standard Apple and Edit menus for you. Other constants define the resource IDs for various dialogs and

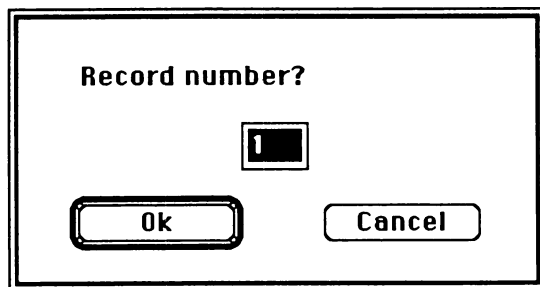


Figure 6.16 The record number dialog of Listing 6.17, DataEntry.

alerts (48–87) along with other miscellaneous items. Lines 59–60 set **KeyReturn** and **KeyEnter** to the key codes for these two keys. As you recall from Chapter 4, a key's code is its position on the keyboard and is not the same as the ASCII code that typing that key produces. Later statements show how to recognize keypresses by their codes.

MaxRec (65) defines how many records the program processes. You can change this value, but keep it small—under 250 probably. Whatever value you choose, the program must be able to hold that many records in memory at once. You can also change **FileName** (66) to write to a different data file on disk.

The constants at lines 68–87 correspond with the resource items that specify labels and edit areas in the dialog. The first set of constants (68–75) limit field lengths. In your own programs, leave enough room to display at least this many characters in the corresponding **EditText** item. The other constants (77–85) identify each of the fields in a database record. Always number these beginning with 1 when designing your own entry forms. They do not refer to positions in dialog item lists, as do similar constants in previous examples. Instead, they simply make the program more readable. For example, **ZipItem** (81) obviously refers to the zip code field, number 5 in the resource file (Listing 6.16, 157–159). Set **MaxField** (87) to the total number of fields in one form, 9 in this example.

Data type **Dollar** (93) stores dollar amounts as 32-bit long integers for typing fields like Payments and Charges in Figure 6.14. **DataRec** (95–106) is a Pascal record that holds one data record as stored on disk. Its fields must correspond with the constants described earlier (68–85). **DataArray** (108) is an array of **MaxRec** records—the in-memory copy of the disk data file. The next type, **DataArrayPtr**, is a pointer to this array, which the program creates on the heap.

The only menu handle is **recordMenu** (114). (MacExtras defines handles for the program's other three menus.) The four **CursHandle** (Cursor Handle) variables (116–119) let the program change cursor styles depending on where you move the mouse pointer. Usually, the pointer is the standard arrow. But inside the dialog window, it changes to a vertical editing bar or to a cross hair for clicking buttons. And, while saving records to disk, it changes to a watch, telling you to wait until that operation completes. Variable **theCursor** (119) holds the handle to the cursor's current shape. The two **Rect** variables, **editArea** and **buttonArea** (121–122), define the areas on screen in which the cursor changes to different patterns.

Pointer **dPtr** (124) is a pointer to the modeless dialog, the entry form in Figure 6.15. Because the dialog operates simultaneously with other program features, its pointer must be a global variable.

Boolean variable **quitRequested** (126) is **TRUE** when you choose the File menu's Quit command. When a second Boolean, **dataDirty** (127), is **TRUE**, it indicates that you made at least one change to a record. The program uses both flags to warn you against quitting without first saving your changes.

Pointer **theData** points to the array of records in memory (128). An index into this array, **dataIndex**, specifies which of those records is now on display in the dialog window. This value equals the record number in the upper right corner.

The final variable, **itemHandles** (136–137), holds handles to each of the edit items in the entry form. This avoids repeatedly calling **GetDItem** to pass strings to and from each of those items and, therefore, speeds displaying many fields at once. The array indices correspond with the constants at lines 77–85. Each constant is an **ItemHandles** array index, which the program uses to edit text for that field.

ResetCursor to ShowRecordNumber (141–190)

ResetCursor (141–148) changes the cursor to its standard arrow shape by calling toolbox procedure **InitCursor**. It then sets global variable **theCursor** to **NIL**—an arbitrary convention that, in this program, indicates the arrow cursor is the one now on display. Other routines check **theCursor** and if it is **NIL**, they know the cursor is an arrow.

InitNewArray (151–163) illustrates one way to quickly initialize large variables, the array of records in this program. Procedure **FillChar** fills any variable with bytes or characters according to this design:

```
FillChar( destination, size, value );
```

Destination can be any variable, even an indexed array. **Size** is the number of bytes that **FillChar** fills with **Value**, which can be any byte value from 0 to 255 or a character. Be extremely careful when using **FillChar**—it does not prevent you from filling beyond the ends of variables in memory! Always use function **Sizeof** as in the example to fill only as many bytes as the destination variable occupies. This limits filling to the byte size of the data array.

Procedure **ShowRecordNumber** (166–190) displays the record number in the upper right corner of the dialog window. It does this by converting global **dataIndex** to a string (185) and inserting that string into the dialog item with a call to **SetIText** at line 188. The **WHILE** loop (186–187) adds leading zeros to this string to display numbers as 002, 010, and so forth. Change the '0' to a blank (' ') if you prefer.

DollarToStr to StrToDollar (193–239)

Two useful tools convert dollar variables (of type **LONGINT**) to and from strings. **DollarToStr** (193–210) uses **NumToString** (204) to convert value **d** to a string of digits. The following **WHILE** loop ensures this string is at least three digits long, and the **Insert** at line 207 adds a decimal point two characters in from the right end of the result. Because of these actions, integer values like 4 and 56 display as 0.04 and 0.56. In other words, a **Dollar**'s unit value is one cent. Notice how the procedure adds a minus sign to strings. Turn lines 202–203 and 208–209 into comments, run the program, and type small negative values like **–5** and **–9** to see why the procedure specially handles negative numbers this way.

StrToDollar (213–239) converts strings back into **Dollar** variables. Sub-

procedure **FindDecimal** (222–229) sets integer **p** to the position of any decimal point in the string. The main procedure calls **FindDecimal** in a **WHILE** loop (233–237) to remove all periods from the string. (Never mind that there should be only one decimal point—somebody might type more than one.) After doing that, **StringToNum** (238) converts the processed string to a long integer value. The best way to type dollar amounts with this system is never to type a decimal point—the way you enter digits on mechanical calculators. (Remember those?)

FieldToDialog to DialogToField (242–259)

Crucial to the success of the data entry form is the ability to convert record fields into **Edit Text** items and then, after editing, to insert changes into records. **FieldToDialog** (242–248) takes a string (**field**) and inserts that string into the **Edit-Text** item of number **itemNum**. Notice that line 247 calls **SetItem**, passing the item's handle from the **itemHandles** array. Without this array, the procedure would have to waste time calling **GetItem** to locate the handle to this item in memory.

Reversing this process, **DialogToField** (251–259) retrieves edited text, setting string variable **field** to new entries by calling **GetItem** (256). Parameter **len** specifies the maximum length of this string. If you type more than this many characters, the procedure cuts the result down to size in the **IF** statement at lines 257–258.

RecToDialog to NewRecord (262–339)

Two procedures call the previous four to convert entire records to and from edit areas in the dialog window. **RecToDialog** (262–288) inserts all fields of the current record into the proper **EditText** items, ready for editing. It directly calls **FieldToDialog** for string fields (274–280), and both **DollarToStr** and **FieldToDialog** for dollar amounts (281–284). The procedure finishes with two miscellaneous jobs, selecting the entire Name field (286) and displaying the record number (287). This highlights the top field (Name) when you move from one record to another.

DialogToRec (291–321) retrieves all edited text from the entry form and inserts that text into the appropriate record fields. For each string field, it calls **DialogToField** and then assigns the result to a field in the record, for example, the city field at line 307. It similarly handles dollar amounts, calling **DialogToField** and then **StrToDollar** for each one (316–319).

When you move to a new record, **RecToDialog** transfers copies of that record's fields into the dialog entry form, ready for editing. When you then click the Next or Previous buttons (or move to a specific record by number), **DialogToRec** extracts any changes you made and inserts them back into the record. Be sure you understand this process—it's much easier than designing programs to directly edit string fields in records. And, because the editing takes place only on copies of the actual data, it's a simple matter to program an Undo command (like the example's Cancel button) that recovers original records. Calling **DialogToRec** is the *only* way data in records change.

NewRecord (324–339) illustrates how to use those two procedures. The program calls it whenever you move from one record to another. First, it saves any changes you made to fields now on display, calling **DialogToRec** to copy edited text into the current record (329). It then changes record number **dataIndex**, limiting it from 1 to **MaxRec** (330–337), and inserts that record's fields into the data entry form for editing (338).

DoNext to ReadFromDisk (342–440)

DoNext and **DoPrevious** (342–357) call **NewRecord** to move to the next or previous record. The program calls these procedures to respond to clicks in the Next and Previous buttons and when you choose those commands from the Record menu.

DoSave (360–404) writes the records in memory to disk. Notice that it first calls **DialogToRec** (377) to save any changes you made to the record now on display. Doing this ensures that all data on disk exactly matches what you see on screen.

Line 380 shows how to change the cursor to a wrist watch, telling you to wait while the procedure writes records to disk. The statements at 382–393 write each record from **theData** array and detect errors by examining function **IoResult**, saving its value in variable **errCode** (383,390). After the disk writes are finished, line 395 turns the cursor back into an arrow. Then, 397–399 call ErrorUnit **IoError** to display an error message or, if there weren't any errors, set **dataDirty** **FALSE**, indicating that records are saved. (Presumably, the program never calls **DoSave** unless **dataDirty** is **TRUE**.)

Notice that lines 401–402 close **dataFile** regardless of whether the previous disk writes succeeded. Because of the compiler option $\{ \$i - \}$ that turns off Turbo's own I/O error checking, line 402 sets **errCode** to **IoResult**, even though the procedure ignores any error here. This fulfills the requirement that programs call **IoResult** after every Input or Output operation when error checking is off.

ReadFromDisk (407–440) does one of two things. If a data file already exists, it reads its records into memory. If the file does not exist, it calls **InitNewArray** (421) to initialize a blank array, which **DoSave** saves to disk when you quit the program after typing records.

DoNumber to KeyDownEvents (443–620)

Other chapters explain much of the programming in this section. **DoNumber** (443–485) displays the dialog window in Figure 6.16, letting you type the number of the next record to edit. It uses the method for entering miscellaneous items that Listing 6.10, **ENTRY.PAS**, demonstrates.

DoClose (488–495) is similar to the procedure in **ApShell** (see Chapter 4) but closes only desk accessories. The program does not allow you to close the data entry dialog window.

You've seen the rest of the programming in this section (498–620) in one form

or another in previous examples. These procedures direct menu commands to the appropriate routines. Notice that line 505 checks **dataDirty**, allowing calls to **DoSave** only if the variable is **TRUE**. Otherwise, it assumes you made no changes and, therefore, refuses to write records to disk. Some programmers insert a message here to tell you that data is already saved. If you want to display such a message, this is where to put it.

One other modified routine from ApShell is **DoEditMenuCommands** (511–530), which adds cut and paste editing features to the entry form fields. Line 516 checks for commands that belong to desk accessories. If they don't, 519 sets **dataDirty** to **TRUE** (assuming that any cutting and pasting changes data). The **CASE** statement (521–527) calls the appropriate Dialog Manager routine (**DlgCut** to **DlgDelete**) or just beeps (522), letting you know that Undo doesn't work here.

DialogEvents to END (623–997)

Skip to the end of the program. As you can see, the Program Engine (978–997) is simple. After initializing (980), it calls **DoSystemTasks** in a **REPEAT** loop that checks for events, passing them to either **MouseDownEvents** or **KeyDownEvents**. This is different from ApShell, which calls **GetNextEvent** to intercept events as they occur. In this case, to handle the modeless dialog, Function **IsProgramEvent** (917–932) returns **TRUE** if **GetNextEvent** also returns **TRUE**, calling **DialogEvents** if function **IsDialogEvent** is **TRUE**.

This action—calling **GetNextEvent** and then, even if the result is **FALSE**, checking with **IsDialogEvent** whether an event belonging to a dialog needs handling—is essential to make the cursor blink. It also lets you type keys like Command-Q and Command-X as well as others to simulate clicking buttons—the way the Return key does the same thing as clicking the Next button in this example.

You can see how this works in procedure **DialogEvents** (623–696), called only if **IsDialogEvent** returns **TRUE** at line 929. After initializing local variable **itemHit** (654), the procedure checks **theEvent** record for a Command keypress (661–665). If it finds a command key, it exits the procedure (665), causing the main program loop to handle the keypress as it normally does. But if it finds a regular keypress, it translates that key by calling sub-procedure **TranslateChar** (634–649), extracting the key code (646) and setting **itemHit** to **Ok** if that code equals either the Return or Enter keys. Therefore, clicking Ok and pressing Return or Enter have the identical effect.

Lines 674–676 further process the dialog event by calling **DialogSelect**, the workhorse that handles all operations inside the dialog, editing text items, sensing mouse clicks, and so on. The reason this section avoids calling **DialogSelect** if **itemHit** equals **NextButton** is to let the program process the Return and Enter keys itself. (**ItemHit** equals **NextButton** at this point only if you had pressed Return or Enter.) If the program didn't take these steps, **DialogSelect** would display a visible character—usually the square box that signifies no specific character—when you press Return.

If **flag** is **TRUE** at line 682, then either you pressed Return (or Enter) or **DialogSelect** set **itemHit** to an item clicked in the window. In these cases, lines 684–694 call **DoNext** or **DoPrevious** to handle clicking those buttons. If you click Cancel, line 690 calls **RecToDialog** to reinsert the fields from the current Pascal record into the dialog edit items, recovering the original fields as they were before editing. Notice also that line 692 assumes that keypresses make changes to records, setting **dataDirty TRUE**.

Changing Cursors

When you move the mouse pointer, its shape changes depending on where it points. Add this feature to programs to help people know what the program expects them to do. Display the vertical bar in areas that accept editing, the mouse pointer in the desktop and menu bar, and a cross hair for buttons. But don't overdo it. Some programs change the cursor so often it becomes impossible to memorize what the various shapes mean.

Procedure **DoSystemTasks** (941) calls **ChangeCursor** (854–914) one time through every main Program Engine cycle. It first checks if the dialog window is the frontmost window (869). If not, it immediately exits, letting desk accessories handle their own cursor changes.

If the dialog window is frontmost, line 876 ensures that it's the current port, saving the original **GrafPort** pointer in local variable **oldPort**. Line 882 sets **Point** record **mouseLocation** to the mouse pointer's coordinate, local to the current window. After that, two **IF** statements (888–893) call **PtInRect** (Point in Rectangle) to check whether this coordinate is inside one of the global rectangles, **editArea** or **buttonArea**. The effect of this is to set **NewCursor** to one of the cursor handles **iBeam** or **cross**, or to **NIL**, signifying the standard arrow pointer.

Then, another **IF** statement (901–907) checks this new setting against the current one stored in global variable **theCursor**. If different, it calls **InitCursor** to display an arrow or **SetCursor** to change to another shape. Notice the double caret dereferencing **NewCursor** to a Cursor data type, which **SetCursor** requires (905).

Initializations (699–851)

SetUpCursors (699–708) initializes the global cursor handles, **iBeam**, **watch**, and **cross**. Calling **GetCursor** with the constants shown here returns handles to these cursor patterns in memory. For absolute safety, you could check whether this works. In other words, it might be better to write:

```
iBeam := GetCursor( IBeamCursor );
IF iBeam = NIL THEN {error};
```

Although this prevents the program from accidentally using an uninitialized shape, because **NIL** indicates a standard arrow, it displays that cursor if another

shape is unavailable. For an example, insert this line between line 706 and 707 and click one of the dialog buttons. Instead of a cross hair, you see an arrow.

```
cross := NIL;
```

Procedure **SetUpDialog** (734–784) initializes and displays the modeless dialog data entry form. It calls **GetNewDialog** (748) to set **dPtr** to the dialog resource in memory and display the dialog window. If that fails, it ends the program immediately by calling **ExitToShell** (750). You might want to improve this by also displaying an error message. The **FOR** loop (757–764) calls **GetDItem** for each of the **Edit-Text** items in the dialog item list, ending the program here also if it detects any errors. It assigns these handles to the **itemHandles** array (763) for procedures **FieldToDialog** and **DialogToField** explained earlier.

Other miscellaneous initialization steps set the **editArea** and **buttonArea** rectangles (772–773) to the areas on screen where the cursor changes shape. Calling **GetDItem** (775) retrieves the Ok button resource to locate that button's position relative to the dialog window. Moving the cursor into this area then changes the cursor to a cross hair.

SetUpArray (787–799) creates the data array as a non-relocatable object on the heap. In previous chapters, you learned that this is rarely a good idea—that objects on the heap should be in relocatable memory blocks. But there are two reasons to break this rule here. First, the data array is large. Its 100 130-byte records occupy 13,000 bytes. (Obviously, it's a mistake to create such a large object as a local variable on the stack.) Second, the array is always in memory while the program runs. For these reasons, it's no help to allow the Memory Manager to move it around. Because the array is always present, a better idea is to place it as low on the heap as possible, below other objects. This avoids fragmenting the heap and lets the program use simple pointer addressing.

The procedure sets this up. It reserves heap space for the array by calling **NewPtr** (792), requesting the **SizeOf** the **DataArray** type in bytes and casting the resulting plain pointer to a **DataArrayPtr** type, assigning this value to **theData**, the pointer to the array. The next line, 793, checks that this worked, examining toolbox function **MemError**. If equal to **NoErr**, then the program is ready to go. Otherwise, there wasn't enough memory available for the array, and the procedure displays an error message (795–796) before ending the program.

Function **QuitConfirmed** (817–851) is the program's deinitialization step. In an earlier example, I said I would explain more fully how to use this function to guarantee against accidentally tossing away your data with the windows. The first job is to check the **dataDirty** flag to decide whether to display an alert warning message at line 837 (see Figure 6.6). If you click the Yes button to save changes before quitting, line 845 calls **DoSave** to write the records in memory to disk. Line 846 sets the **quitRequested** flag, indicating whether that disk operation worked. If not, the program must not end. Closely examine **QuitConfirmed** and be sure you understand how it works. Its design is appropriate in any program that writes data to disk files.

DIALOG TOOLS

The examples in this chapter all make use of tools in DialogUnit. This section details the unit's individual routines which you'll find handy when designing your own dialogs.

Type in Listing 6.18 and save as DIALOGUNIT.PAS. Compile to a disk file. You cannot run the unit—it requires a *host* program as do all units. One change you might have to make is in line 1. If your disk and folder names are different, insert them here in place of Programs and Units.F.

Listing 6.18. DIALOGUNIT.PAS

```

1: {$O Programs:Units.F: }      { Send compiled code to here }
2: {$U-}                        { Turn off standard library units }
3:
4:
5: UNIT DialogUnit( 130 );
6:
7: (*
8:
9:  * PURPOSE : Dialog support routines
10: * SYSTEM  : Macintosh / Turbo Pascal
11: * AUTHOR  : Tom Swan
12:
13: *)
14:
15:
16: INTERFACE                    { Items visible to a host program }
17:
18:
19:     USES
20:
21:         Memtypes, QuickDraw, OSIntf, ToolIntf, PackIntf;
22:
23:
24:     TYPE
25:
26:         ButtonRec =
27:             RECORD
28:                 firstButton : INTEGER;    { First radio button item number }
29:                 lastButton  : INTEGER;    { Last item number }
30:                 selection   : INTEGER;    { Current button (-1 if none) }
31:             END; { ButtonRec }
32:
33:         CheckSet = SET OF 0 .. 255;      { Sets of checked boxes }
34:
35:         ChecksRecord =
36:             RECORD
37:                 firstCheck  : INTEGER;    { First item number in check list }
38:                 lastCheck   : INTEGER;    { Last item number }
39:                 selections  : CheckSet;   { Set of currently checked boxes }
40:             END; { ChecksRecord }
41:
42:
43: PROCEDURE ToggleCheck( ch : Handle );
44:
45: FUNCTION CheckOn( ch : Handle ) : BOOLEAN;
46:

```

```

47: PROCEDURE OutlineOk( dPtr : DialogPtr );
48:
49: FUNCTION HCenter( width : INTEGER ) : INTEGER;
50:
51: FUNCTION VCenter( height : INTEGER ) : INTEGER;
52:
53: FUNCTION MakeFileName( VAR reply : SFReply;
54:                        prompt : Str255; fileName : Str255 ) : BOOLEAN;
55:
56: FUNCTION GetFileName( VAR reply : SFReply; fileKind : OSType ) : BOOLEAN;
57:
58: PROCEDURE PushButton( dp : DialogPtr; VAR buttons : ButtonRec;
59:                      itemHit : INTEGER );
60:
61: PROCEDURE InitButtons( dp : DialogPtr; VAR buttons : ButtonRec );
62:
63: PROCEDURE CheckBox( dp : DialogPtr; VAR checks : ChecksRecord;
64:                    itemHit : INTEGER );
65:
66: PROCEDURE InitChecks( dp : DialogPtr; VAR checks : ChecksRecord );
67:
68:
69:
70: IMPLEMENTATION      { Items not visible to a host program }
71:
72:
73: FUNCTION InRange2( n, min, max : INTEGER ) : BOOLEAN;
74:
75: { Returns TRUE if min <= n <= max (borrowed from MacExtras.PAS).
76:   Remove if you use that unit and DialogUnit together. }
77:
78: BEGIN
79:     InRange2 := ( min <= n ) AND ( n <= max )
80: END; { InRange2 }
81:
82:
83: PROCEDURE ToggleCheck;
84:
85: { Toggle check mark or radio button control on/off.
86:   NOTE: ch is a plain handle but it's up to you to make certain that it
87:   actually addresses a checkmark or radio button control. }
88:
89: VAR
90:
91:     n : INTEGER;
92:
93: BEGIN
94:     n := GetCtlValue( ControlHandle( ch ) ); { Get current setting }
95:     IF n = 0
96:     THEN n := 1      { Toggle on (1) / off (0) }
97:     ELSE n := 0;
98:     SetCtlValue( ControlHandle( ch ), n )    { Change setting }
99: END; { ToggleCheck }
100:
101:
102: FUNCTION CheckOn;
103:
104: { TRUE if the check mark or radio button with this handle is on. It's
105:   up to you to make sure that ch really addresses a control. The control
106:   is on if it has a non zero value; it's off only if it equals zero. }
107:
108: BEGIN
109:     CheckOn := GetCtlValue( ControlHandle( ch ) ) <> 0
110: END; { CheckOn }
111:
112:

```

(continued)

```

113: PROCEDURE OutlineOk;
114:
115: { Draw bold outline around the Ok button (or some other button) in the
116:   dialog addressed by dPtr. Assumes the Ok button is the first item
117:   in the dialog's item list. The dialog window does not have to be
118:   the current port. Note: changes pen size. }
119:
120:   VAR
121:
122:       oldPort      : GrafPtr;
123:       itemType     : INTEGER;
124:       itemHandle   : Handle;
125:       itemRect     : Rect;
126:
127:   BEGIN
128:       GetPort( oldPort );
129:       SetPort( dPtr );
130:       GetDItem( dPtr, 1, itemType, itemHandle, itemRect );
131:       IF itemHandle <> NIL THEN
132:           BEGIN
133:               PenSize( 3, 3 );
134:               InsetRect( itemRect, -4, -4 );
135:               FrameRoundRect( itemRect, 16, 16 )
136:           END; { if }
137:       SetPort( oldPort )
138:   END; { OutlineOk }
139:
140:
141: FUNCTION HCenter;
142:
143: { Return horizontal coordinate value for centering a dialog
144:   window of this width between the left and right screen borders }
145:
146:   BEGIN
147:       HCenter := ( screenBits.bounds.right - width ) DIV 2
148:   END; { HCenter }
149:
150:
151: FUNCTION VCenter;
152:
153: { Return vertical coordinate value for centering a dialog
154:   window of this height between the top and bottom screen borders }
155:
156:   BEGIN
157:       VCenter := ( screenBits.bounds.bottom - height ) DIV 2
158:   END; { VCenter }
159:
160:
161: FUNCTION MakeFileName;
162:
163:   CONST
164:
165:       DlogWidth   = 304; { Standard file "put" dialog box width }
166:       DlogHeight  = 184; { and height }
167:
168:   VAR
169:
170:       where      : Point;
171:
172:   BEGIN
173:       SetPt( where, HCenter( DlogWidth ), VCenter( DlogHeight ) );
174:       SFPutFile( where, prompt, fileName, NIL, reply );
175:       MakeFileName := reply.good
176:   END; { MakeFileName }
177:
178:

```

```

179: FUNCTION GetFileName;
180:
181: { TRUE if someone selects an file name of type fileKind.  If so, reply
182:   holds its file name and volume information. }
183:
184:   CONST
185:
186:       DlogWidth   = 348;   { Standard file "get" dialog box width }
187:       DlogHeight  = 200;   { and height }
188:
189:   VAR
190:
191:       typeList : SFTypelist;
192:       where    : Point;
193:
194:   BEGIN
195:       SetPt( where, HCenter( DlogWidth ), VCenter( DlogHeight ) );
196:       typeList[ 0 ] := fileKind;   { Allow only these files }
197:       SFGetFile( where, '', NIL, 1, typeList, NIL, reply );
198:       GetFileName := reply.good    { True if Open clicked }
199:   END; { GetApplName }
200:
201:
202: PROCEDURE PushButton;
203:
204: { Push the radio button number itemHit and save in buttons rec.  Turn
205:   off any previously-selected button and turn on the new one. }
206:
207:   VAR
208:
209:       itemType    : INTEGER;      { Ignored }
210:       itemHandle   : Handle;       { Handle to button control }
211:       itemRect     : Rect;         { Ignored }
212:
213:   BEGIN
214:       WITH buttons DO
215:           IF InRange2( itemHit, firstButton, lastButton ) THEN
216:               BEGIN
217:
218:                   { Turn off current button (if there is one) }
219:
220:                   IF InRange2( selection, firstButton, lastButton ) THEN
221:                       BEGIN
222:                           GetDItem( dp, selection,
223:                                     itemType, itemHandle, itemRect );
224:                           SetCtlValue( ControlHandle( itemHandle ), 0 )
225:                       END; { if }
226:
227:                   { Turn on new button }
228:
229:                   GetDItem( dp, itemHit, itemType, itemHandle, itemRect );
230:                   SetCtlValue( ControlHandle( itemHandle ), 1 );
231:
232:                   selection := itemHit    { Remember current button }
233:
234:               END { with / if / if }
235:           END; { PushButton }
236:
237:
238: PROCEDURE InitButtons;
239:
240: { Initialize radio button display to match buttons record info.  Call
241:   this procedure after displaying the dialog window to turn on the button
242:   matching the selection in the buttons record. }
243:
244:   BEGIN
245:       PushButton( dp, buttons, buttons.selection )
246:   END; { InitButtons }

```

(continued)


```

247:
248:
249: PROCEDURE CheckBox;
250:
251: { Toggle check itemHit on or off and save current value in checks }
252:
253:   VAR
254:
255:       itemType      : INTEGER;      { Ignored }
256:       itemHandle     : Handle;       { Handle to check box control }
257:       itemRect       : Rect;         { Ignored }
258:
259: BEGIN
260:   WITH checks DO
261:     IF ( firstCheck <= itemHit ) AND ( itemHit <= lastCheck ) THEN
262:       BEGIN
263:         GetDItem( dp, itemHit, itemType, itemHandle, itemRect );
264:         ToggleCheck( itemHandle );
265:         IF CheckOn( itemHandle )
266:           THEN selections := selections + [ itemHit ]
267:           ELSE selections := selections - [ itemHit ]
268:         END { if }
269:       END; { CheckBox }
270:
271:
272: PROCEDURE InitChecks;
273:
274: { Set up check marks in boxes to correspond with information
275:   in this checks record }
276:
277:   VAR
278:
279:       checkNumber : INTEGER;
280:
281: BEGIN
282:   WITH checks DO
283:     FOR checkNumber := firstCheck TO lastCheck DO
284:       IF checkNumber IN selections
285:         THEN CheckBox( dp, checks, checkNumber )
286:       END; { InitChecks }
287:
288:
289:
290: END. { DialogUnit }

```

DialogUnit Play-by-Play

The unit uses five others—all of which are included in the Turbo Pascal file, ready to use. Your host program must use at least these five listed at line 21 along with `DialogUnit`. Listings 6.8 (RADIO.PAS) and 6.12 (OPTIONS.PAS) show how to use the unit's data types, **ButtonRec**, **CheckSet**, and **ChecksRecord** (26–40), to organize radio buttons and check boxes in dialogs. The following notes describe each of `DialogUnit`'s procedures and functions.

```
FUNCTION InRange2( n, min, max : INTEGER ) :  
    BOOLEAN;
```

This function is identical to **InRange** in the MacExtras unit (see Chapter 4). It's included here to keep examples in this chapter simple. (Programs that use MacExtras must have pull-down menus and respect event-driven programming rules.) If you plan to use DialogUnit and MacExtras together, remove lines 73–80 and replace all calls to **InRange2** with **InRange**.

```
PROCEDURE ToggleCheck( ch : Handle );
```

Pass a control handle **ch** to a radio button or check box to toggle its setting on or off. You get the handle by calling **GetDItem** with a **Handle** variable as the fourth parameter. **ToggleCheck** (83–99) casts your handle to type **ControlHandle** in calls to **GetCtlValue** (94) and **SetCtlValue** (98), setting the button or box to 1 to turn it on or to 0 to turn it off.

```
FUNCTION CheckOn( ch : Handle ) : BOOLEAN;
```

Call **CheckOn** (102–110) with any radio or check box handle to check whether this control is now on (**TRUE**) or off (**FALSE**).

```
PROCEDURE OutlineOk( dPtr : DialogPtr );
```

Pass a dialog pointer (**dPtr**) to **OutlineOk** (113–138) to draw a bold outline around the first control in the dialog's item list, which should be a button. (See Figure 6.16 for an example. The button on the left has a heavy border, drawn by **OutlineOk**.) Be sure that the first item in the dialog list (DITL) is a button. If it isn't, **OutlineOk** might produce a strange result or not work at all. The button can have any title—it doesn't have to be "Ok."

One problem when outlining buttons this way is that if another window should cover the button and then move aside, the dialog manager will redraw the outline with a normal border. Because your program has no way to sense the situation, it cannot redraw the outline in bold. This problem is rare, though, and not worth spending too much time solving. It can never happen to modal dialogs.

```
FUNCTION HCenter( width : INTEGER ) : INTEGER;  
FUNCTION VCenter( height : INTEGER ) : INTEGER;
```

HCenter and **VCenter** (141–148) work together to help center dialog windows in the Macintosh display. Pass the width of the dialog window to **HCenter** and the height to **VCenter**. Together, the functions return the global (H,V) coordinate where, if you place the dialog window's upper left corner, the window is exactly

centered on screen. The next two functions call **HCenter** and **VCenter** to center the standard file dialogs even on Macintosh models with large displays.

```
FUNCTION MakeFileName( VAR reply : SFReply;  
    prompt : Str255; fileName : Str255 ) : BOOLEAN;
```

Call **MakeFileName** to choose new file names, usually in response to the File menu's Save as command. If the function returns **TRUE**, then **reply** specifies the file name and volume where the program should write its data. This file may or may not exist. If it does, overwrite it—the dialog manager has already requested permission to remove the old file.

The function calls **SetPt** (173) to position **Point** record **where** according to the values that **HCenter** and **VCenter** return. Constants **DlogWidth** and **DlogHeight** equal the standard file dialog sizes. **SFPutFile** (174) displays the dialog window (see Figure 6.2), lets you eject disks, open folders, and so on, and returns the **reply** record filled with the new name or with field **good** equal to **FALSE** if you click the Cancel button. It also checks whether files already exist and gives you the chance to change your mind about erasing them. As long as **MakeFileName** returns **TRUE**, you can write to the disk file name in **reply**.

```
FUNCTION GetFileName( VAR reply : SFReply;  
    fileKind : OSType ) : BOOLEAN;
```

Use **GetFileName** to choose existing files from disk, usually in response to the File menu's Open command. It displays the standard dialog window in Figure 6.1 and lets you eject disks, open folders, and double-click names or click the other dialog buttons. If it returns **TRUE**, open the file name in **reply**.

OSType parameter **fileKind** (196) specifies the kind of file you want to see in the dialog directory window. Pass 'TEXT' for text files, 'APPL' for applications, and others to limit files to specific types. See Table 6.1 for a list of other file types you can use.

```
PROCEDURE PushButton( dp : DialogPtr;  
    VAR buttons : ButtonRec; itemHit : INTEGER );
```

PushButton (202–235) takes three parameters, a dialog pointer (**dp**), a **ButtonRec** record (**buttons**), and an integer (**itemHit**). Call the procedure after **ModalDialog** indicates a hit in a radio button inside the dialog window. Pass the **itemHit** number that **ModalDialog** returns and **PushButton** turns that button on and the others off.

The **buttons** record specifies the first and last button numbers and the current selection, the button now on. See Listing 6.8 (RADIO.PAS) for an example of how to use **PushButton**.

```
PROCEDURE InitButtons( dp : DialogPtr;
  VAR buttons : ButtonRec );
```

Call **InitButtons** (238–246) after loading and displaying a dialog window, usually with **GetNewDialog** as many examples in this chapter demonstrate. Pass the dialog pointer (**dp**) that **GetNewDialog** returns and a **ButtonRec** record (**buttons**) with its parameters set as explained in the play-by-play notes to Listing 6.8 (RADIO.PAS). **InitButtons** darkens the currently punched button, leaving the others blank. If you don't want to punch any buttons, set **buttons.selection** to **-1**.

```
PROCEDURE CheckBox( dp : DialogPtr;
  VAR checks : ChecksRecord; itemHit : INTEGER );
```

CheckBox (249–269) is similar to **PushButton**. It adds or subtracts a check mark from the check box that **itemHit** specifies. Call it after **ModalDialog** indicates a click in a check box belonging to a dialog window.

Variable **checks** holds the first and last check item number and a set of checked boxes in field **selections**, of type **CheckSet**. After calling **CheckBox**, **selections** holds all the boxes that now have check marks. See the play-by-play to Listing 6.12 (OPTIONS.PAS) for more information about using this routine.

```
PROCEDURE InitChecks( dp : DialogPtr;
  VAR checks : ChecksRecord );
```

Call **InitChecks** (272–286) after loading and displaying a dialog, usually with **GetNewDialog**. Pass the dialog pointer (**dp**) and a **checks** record to add check marks to all boxes that the **checks.selections** set specifies. After calling **InitChecks**, the dialog display matches the parameters in the checks record. See the play-by-play to Listing 6.12 (OPTIONS.PAS) for additional details.

Units as Software Tools

Because you create them separately from the main program, units reduce compiling time while isolating common routines and data that programs share. They have two primary uses. In a large program, you insert tested procedures and functions into units to avoid recompiling those same routines over and over. Or you can build a library of units with routines for many different kinds of programs.

This chapter presents three units of the second variety. *Transfer* lets one program transfer to another. *IconUnit* contains tools for manipulating icon images inside windows. And *ImageUnit* develops tools for using the *ImageWriter*'s native printing abilities. At the end of the chapter is *MacLister*, a program lister that uses most of the units in this book.

DEVELOPING A SOFTWARE LIBRARY

After creating a unit, there are several ways to use it. Normally, you compile it to disk, creating a code file to which other programs refer, the method most of the examples in this book use. If you compile units to memory instead, programs can use them only while their text windows remain open. Although this is helpful for testing minor changes, you normally compile units to disk code files to make them available to other programs.

Up to now, programs expect to find their units on volume *Programs* in folder *Units.F*. To use such a unit requires placing a Unit compiler directive ahead of the program's **USES** declaration. For example, many examples in this book include the line:

```
{ $U Programs:Units.F:MacExtras }
```

This opens the *Unit.F* folder, looking for the code file *MacExtras*. You need to include this step only for the units you create and compile to disk. Others, such as *QuickDraw* and *ToolIntf*, don't require a Unit directive—they're old friends to

the compiler and, in fact, are stored inside the Turbo Pascal file. It takes the compiler less time to use such units than it does to locate and open disk files containing compiled unit code.

To do this with your own units, you can install them inside the Turbo Compiler, reshaping it to understand new commands and data types. For example, if you install the MacExtras unit into the compiler, you can then add it to a **USES** declaration without also including a Unit compiler directive to find the unit code file on disk. In fact, then you no longer need the unit code file at all. Although attractive for that reason, installing units in the compiler is not something to do haphazardly. For best results, follow these suggestions.

- Install only well-tested units in near final form. Remember that moving units in and out of the compiler is itself a time-consuming operation.
- Keep a list of unit numbers to avoid conflicts. The best place for this list is in a file, perhaps named Turbo Units, on the same volume as the compiler.
- Keep a printout of at least the unit's interface section, the portion visible to programs. If you have room, keep a reference disk copy of this same text along with the compiler.

Installing Units in the Compiler

Installing a unit into Turbo Pascal is easy. Open the utility program UnitMover on your Turbo disk. Your screen should resemble Figure 7.1. In the top left box are the units now installed in the compiler. Initially, the top right box is empty. For practice, follow these steps to install the MacExtras unit from Chapter 4, assuming of course that you previously compiled that unit to disk.

1. Click the Open button (labeled Close in the figure) below the top right box. Locate and open the MacExtras code file.
2. Select MacExtras by clicking its name in the top right box. This highlights the name and activates the Copy button. Your screen should now match the figure.
3. Click Copy to install MacExtras.

To install another unit, close the top right box and repeat these three steps. To remove a unit, select its name and click the Remove button. You cannot remove all run-time units, meaning those required to run Pascal programs. And you cannot remove units that others use.

After installing MacExtras (and others), quit UnitMover and open Turbo. Edit APSHELL.PAS (Listing 4.1) and remove the dollar sign from line 17, turning the Unit compiler directive into a plain comment. (You could also remove the line but would then have to retype it to go back to reading the unit from a disk code file.) Compile ApShell. Turbo now uses the MacExtras unit you installed.

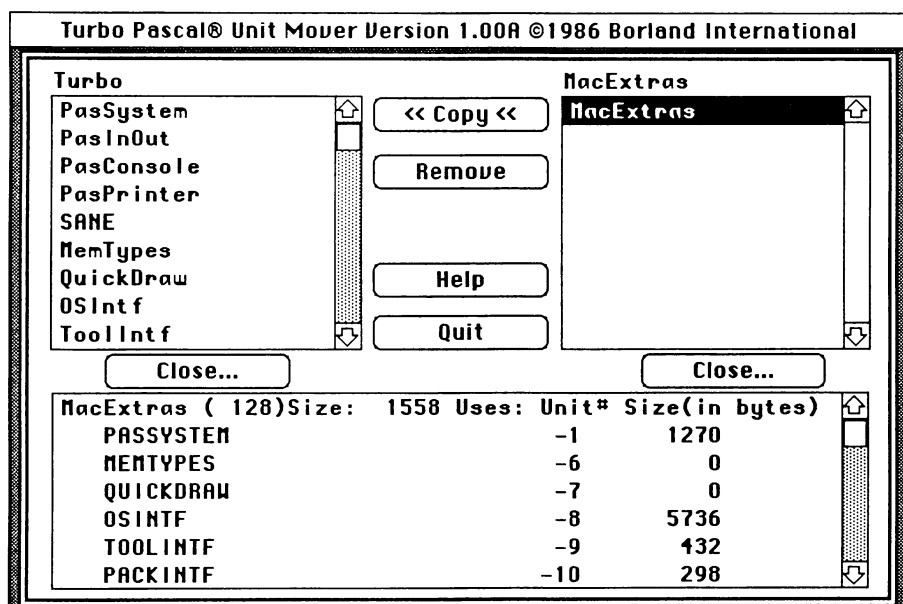


Figure 7.1 Turbo's UnitMover utility program installs and removes compiled units from the compiler. This copy of the program's display shows the MacExtras unit from Chapter 4 about to be copied into the compiler. The information window at bottom displays additional facts about selected units.

When making temporary changes to installed units, you do not have to remove them from the compiler. For example, open the MACEXTRAS.PAS text file and turn line 391 into a comment, temporarily removing the unit's ability to display the "About program" box. Also open APSHELL.PAS. Compile MacExtras to memory and then compile and run ApShell. Try using the About command—it won't operate, proving that Turbo uses the temporary in-memory code in place of the installed unit. Now close the MacExtras text file (you probably do not want to save the change you just made). Compile ApShell and run. The About command should again work. Remember this trick when modifying library units—you can temporarily compile them to memory, test your changes and then later compile to disk before installing the final version into the compiler.

The UnitMover Information Window

At the bottom of the UnitMover display (Figure 7.1) is a description of the units you select in the top left and right windows. To see descriptions of all installed units, click and drag the mouse pointer over their names, highlighting them all.

Then use the vertical scroll bar near the bottom right corner to scroll the information window up and down.

Text in the window tells you the unit name, its number in parentheses, and its size in bytes. Indented below the selected unit name are the other units that this one uses. For example, Figure 7.1 tells you that MacExtras uses the six other units listed in the bottom window (PASSYSTEM to PACKINTF). A program that uses MacExtras must use these units also.

The unit number column, second from the right in the bottom window, lists unit numbers. Notice that system units, the ones that come with your Turbo system, have negative numbers. Your own units have unique positive numbers of any value from 1 to 32767 for all units that programs use. All units in this book have unique numbers starting with 128. (See Listing 7.4, for example, line 5. The unit number is 134.) You may change any of these numbers if they conflict with other units in your library.

The final column in the UnitMover information window lists the byte size of each unit. Some lengths are zero, a normal condition. Such units contain only declarations for objects and routines in the Macintosh toolbox—they contain no code. Using them adds their definitions to a program, but does not increase its compiled size.

TRANSFER TOOLS

Most commercial programs, Turbo Pascal included, can transfer to another program. The next example adds this same ability to your own Pascal projects. First type in Listing 7.1, and save as TRANSFER.PAS. Compile the unit to a disk code file. After the play-by-play description is an example program that uses Transfer to run another application.

Listing 7.1. TRANSFER.PAS

```

1: {$O Programs:Units.F: }      { Send compiled code to here }
2: {$U-}                        { Turn off standard library units }
3:
4:
5: UNIT Transfer( 129 );
6:
7: (*
8:
9:  * PURPOSE : Transfer from one program to another
10:  * SYSTEM  : Macintosh / Turbo Pascal
11:  * AUTHOR   : Tom Swan
12:
13:  *)
14:
15:
16: INTERFACE                    { Items visible to a host program }
17:
18:

```

```

19:  USES
20:
21:      Memtypes, QuickDraw, OSIntf, ToolIntf, PackIntf;
22:
23:
24: FUNCTION GetApplName( VAR reply : SFReply ) : BOOLEAN;
25:
26: PROCEDURE RunProgram( reply : SFReply );
27:
28:
29:
30: IMPLEMENTATION      { Items not visible to a host program }
31:
32:
33:  TYPE
34:
35:      TransferPtr = ^TransferRec;
36:
37:      TransferRec =
38:          RECORD
39:              fNamePtr : ^Str255;      { File name }
40:              config   : INTEGER      { 0=normal screen & sound buffers * }
41:          END; { TransferRec }
42:
43:
44: { * Note: set config > 0 to allocate alternate sound buffer; set
45:   it to < 0 to allocate alternate sound and alternate screen buffers.
46:   But avoid doing this unless absolutely necessary--it may not work
47:   on all Macintosh models. Normally set config to 0. }
48:
49:
50: { The following in-line procedure executes the 68000 instructions:
51:
52:      MOVE.L    (SP)+, A0    ; Move param tp to A0
53:      _Launch   ; Execute Launch trap  }
54:
55:
56: PROCEDURE Launch( tp : TransferPtr );
57:     INLINE $205F, $A9F2;
58:
59:
60: FUNCTION GetApplName;
61:
62: { TRUE if someone selects an application file name. If so, reply
63:   holds its file name and volume information. }
64:
65:  VAR
66:
67:      typeList : SFTYPEList;
68:      where    : Point;
69:
70:  BEGIN
71:      SetPt( where, 85, 100 );      { Standard file dialog location }
72:      typeList[ 0 ] := 'APPL';      { Allow only application files }
73:      SFGetFile( where, '', NIL, 1, typeList, NIL, reply ); { Do dialog }
74:      GetApplName := reply.good     { True if Open clicked }
75:  END; { GetApplName }
76:
77:
78: PROCEDURE RunProgram;
79:
80: { Run program specified in reply }
81:
82:  VAR
83:
84:      tRec : TransferRec;
85:

```

(continued)

```

86:   BEGIN
87:     IF SetVol( NIL, reply.vRefNum ) = noErr THEN
88:       BEGIN
89:         WITH tRec DO
90:           BEGIN
91:             fNamePtr := @reply.fname; { Assign file name }
92:             config := 0                { Main sound & screen buffers }
93:           END; { with }
94:           Launch( @tRec )    { Run (launch) the program }
95:         END { if }
96:       END; { RunProgram }
97:
98:
99: END. { Transfer unit }
100:

```

Transfer Play-by-Play

Not a large unit, Transfer contains only two routines that programs can use. The first, **GetApplName**, displays the familiar file dialog, prompting for the name of an application to run, and limiting your choices to files of type APPL. Pass it an **SFReply** record. If the function returns **TRUE**, call the second routine, **RunProgram**, passing the same record as a parameter. For example, you might use this simple **IF** statement:

```

VAR reply : SFReply;

IF GetApplName( reply )
  THEN RunProgram( reply );

```

In many cases, that's all you need to do to respond to a transfer command from a pull-down menu. To run a specific program and not use the standard file dialog window, create your own **SFReply** record and pass it to **RunProgram**. Using this idea, the following runs a program MYPROG on the current volume:

```

VAR reply : SFReply;

reply.vRefNum := 0;
reply.fname := 'MYPROG';
RunProgram( reply );

```

For a successful transfer, you must observe a few rules. Breaking any of these will cause serious problems and may require you to reboot to recover.

- Never transfer from a program running directly in Turbo Pascal's editor. For transfer to work, you must compile the program to disk and run it from the Finder (or first transfer to it from another application).

- Close all open files before transferring. If you don't do this, other programs may be unable to use those files until you later reboot.
- Be certain that the application to which you transfer exists before calling **RunProgram**. If you use **GetApplName** as described earlier, you'll never have a problem. But if you create your own **reply** record, it's up to you to ensure that the application is on the volume you specify.

The following notes describe how Transfer's two routines operate. For reference, their parameter lists are repeated here.

```
FUNCTION GetApplName( VAR reply : SReply ) :  
  BOOLEAN;
```

GetApplName (60–75) works by calling **SGetFile** (73), passing the upper left corner of the dialog window in local variable **where** along with several other parameters. Change the null string to add a message such as “Select application to run.” The first **NIL** specifies the standard filter function, meaning here that the single string in **typeList**, ‘APPL’, is the only limiting factor in the files shown. (There isn't room here to cover adding your own filter functions—consult *Inside Macintosh* for details.) The **1** indicates that **typeList** holds only this single string. The second **NIL** tells the system to use the standard event handlers for the dialog window. And the last parameter, **reply**, is the result—containing the selected file specifications.

```
PROCEDURE RunProgram( reply : SReply );
```

RunProgram (78–96) transfers control to the file specified in **reply** fields **vRefNum** and **fname**. It ignores other fields in this record. The procedure makes use of a transfer record (type **TransferRec**), which contains two fields (37–41). The first field (**fNamePtr**) is a pointer to a string that contains the name of the program to run. The second field (**config**) is an integer, usually zero. If this value is negative, then the operating system allocates space for alternate sound and screen memory buffers. If positive, it allocates space for an alternate sound buffer only. A zero value allocates no alternate buffers. Beware that these features do not operate on all Macintosh models. A few programs use these extra memory buffers, but most don't. You'll rarely set **config** to anything but zero.

Line 87 sets the default volume to the one you specify in record **reply**. If this works, it then assigns the address of the file name (91) and sets **config** to zero. It then calls **Launch** (94), passing the address of the transfer record.

Procedure **Launch** is an in-line machine language routine (50–57). Although declared as a procedure, it is more of a simple definition because Turbo inserts its code—the hex values in line 57—directly into the program wherever you place its name. In the Pascal program, it appears as though you are calling a procedure (see

line 94), but you are really telling Pascal to insert the in-line programming at this point. Here, the machine language routine moves the address of the transfer record into register A0 and then executes the operating system **Launch** trap. (Don't be concerned if you are unfamiliar with these 68000-microprocessor terms—you don't need to understand them to use **Launch**.)

LET'S DO LAUNCH

A simple program demonstrates how to use the Transfer unit. Type in Listing 7.2, save as LAUNCHER.R, and compile with RMaker to create the program's resource file. Then type in Listing 7.3, save as LAUNCHER.PAS, and compile with Turbo to a disk code file. Copy portions of APSHELL.PAS from Chapter 4 where the listing indicates. Before running the program, observe the following caution.

Do not run Launcher from inside the Turbo editor or you may have to reboot from the resulting system crash. You must either quit to the Finder and open Launcher, or use Turbo's Transfer command to transfer to the program. You can then transfer back to Turbo or to any other application.

Listing 7.2. LAUNCHER.R

```

1: *-----*
2: * Launcher.PAS resources -- Compile with RMaker *
3: *-----*
4:
5: Programs:Launcher.F:Launcher.RSRC    ;; Send output to here
6:
7:
8: *-----*
9: * About box string list *
10: *-----*
11:
12: TYPE STR#                                ;; String list resource
13:     ,1 (32)                             ;; ID and attribute (purgeable)
14: 6                                         ;; Number of strings that follow
15: Program Launcher                        ;; Program name
16: by Tom Swan                            ;; Author
17: Version 1.00                           ;; Version number
18: (C) 1987 by Swan Software               ;; Copyright notice
19: P. O. Box 206, Lititz, PA 17543        ;; Address
20: (717)-627-1911                          ;; Telephone
21:
22:
23: *-----*
24: * The Apple Info menu *
25: *-----*
26:
27: TYPE MENU
28:     ,1                                   ;; Menu ID number to use in program
29: \14                                     ;; Bitten-apple graphics symbol
30:     About Launcher...                   ;; The command as shown in menu
31:     (-                                  ;; Divider line between command and DAs
32:
33:

```

```

34: *-----*
35: * The File menu *
36: *-----*
37:
38: TYPE MENU
39:     ,2                ;; Menu ID number to use in program
40: File                 ;; Menu title as shown in menu bar
41:     New /N
42:     (Close
43:     (-
44:     Transfer /T
45:     Quit /Q
46:
47:
48: *-----*
49: * The Edit menu *
50: *-----*
51:
52: TYPE MENU
53:     ,3
54: Edit
55:     (Undo /Z
56:     (-
57:     (Cut /X
58:     (Copy /C
59:     (Paste /V
60:     (Clear
61:
62:
63: *-----*
64: * Window template *
65: *-----*
66:
67: TYPE WIND
68:     ,1 (32)           ;; ID number and attribute (purgeable)
69: Untitled              ;; Window title
70: 46 7 328 502         ;; top, left, bottom, right coordinates
71: Visible GoAway       ;; Visible window with close button
72: 8                    ;; Standard doc window with grow & zoom boxes
73: 0                    ;; Window reference (none)
74:
75:
76:
77: * END

```

Listing 7.3. LAUNCHER.PAS

```

1: {!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
2:
3: WARNING : Do not use the File menu Transfer command when compiling and
4: running Launcher directly from Turbo Pascal. The command works properly
5: only if you compile the program to disk and transfer to it or run it from
6: the Finder.
7:
8: {!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
9:
10:
11: {$O Programs:Launcher.F: }           { Send compiled code to here }
12: {$R Programs:Launcher.F:Launcher.Rsrc} { Use this compiled resource file }
13: {$U-}                                { Turn off standard library units }
14:
15:

```

(continued)

```

16: PROGRAM Launcher;
17:
18: (*
19:
20:  * PURPOSE : Demonstrate using Transfer unit
21:  * SYSTEM  : Macintosh / Turbo Pascal
22:  * AUTHOR  : Tom Swan
23:
24: *)
25:
26:
27: {$U Programs:Units.F:MacExtras } { Open these library unit files }
28: {$U Programs:Units.F:Transfer }
29:
30:
31:
32:   USES
33:
34:       Memtypes, QuickDraw, OSIntf, ToolIntf, PackIntf,
35:       MacExtras, Transfer;
36:
37:
38:   CONST
39:
40:       FileID      = 2;      { File menu Resource ID and commands }
41:       NewCmd       = 1;
42:       CloseCmd     = 2;
43:       {-----}
44:       TransferCmd  = 4;
45:       QuitCmd      = 5;
46:
47:       WindowID     = 1;      { Window resource ID }
48:
49:
50:
51:   VAR
52:
53:       wRec         : WindowRecord;      { Program's window data record }
54:       wPtr         : WindowPtr;         { Pointer to above wRec }
55:
56:       quitRequested : BOOLEAN;          { TRUE if quitting }
57:       windowOpen    : BOOLEAN;          { TRUE only if window is open }
58:
59:
60:
61: << INSERT LINES 61-121 FROM APSHELL.PAS >>
62:
63:
64: PROCEDURE DoTransfer;
65:
66: { Respond to File menu Transfer command }
67:
68:   VAR
69:
70:       reply : SFReply;
71:
72:   BEGIN
73:       IF GetApplName( reply )
74:       THEN RunProgram( reply )
75:       END; { DoTransfer }
76:
77:
78: PROCEDURE DoFileMenuCommands( cmdNumber : INTEGER );
79:
80: { Execute command in the File menu }
81:

```

```

82: BEGIN
83:     CASE cmdNumber OF
84:         NewCmd       : DoNew;
85:         CloseCmd      : DoClose;
86:         TransferCmd   : DoTransfer;
87:         QuitCmd       : quitRequested := TRUE
88:     END { case }
89: END; { DoFileMenuCommands }
90:
91:
92: << INSERT LINES 137-431 FROM APSHELL.PAS >>
93:
94:
95: BEGIN
96:
97:     Initialize;
98:
99:     REPEAT
100:
101:         DoSystemTasks;
102:
103:         IF GetNextEvent( everyEvent, theEvent ) THEN
104:
105:             CASE theEvent.what OF
106:
107:                 MouseDown      : MouseDownEvents;
108:                 KeyDown        : KeyDownEvents;
109:                 AutoKey         : { ignored };
110:                 UpdateEvt       : UpdateEvents;
111:                 ActivateEvt     : ActivateEvents
112:
113:             END { case }
114:
115:         UNTIL QuitConfirmed
116:
117:     END.

```

Launcher Play-by-Play

LAUNCHER.R (1-77)

Except for the File menu (38-45), Launcher's resource file is identical to Ap-Shell's. Most programmers place Transfer just above the File menu's Quit command as shown here, but you could put in anywhere you like.

LAUNCHER.PAS (1-117)

Launcher is nearly identical to ApShell. Procedure **DoTransfer** (64-75) responds to choosing the Transfer command from the File menu. It simply calls **GetAppName** (73) to display the standard file dialog, letting you choose an application name and, if the function returns **TRUE**, calls **RunProgram** to transfer to that program. Procedure **DoFileMenuCommands** (78-89) adds the transfer command to its usual list of **CASE** selectors (86).

When transferring from your own programs, remember to close all open files before calling **RunProgram**. You could add a call in **DoTransfer** to your program's

file closing routine, or set a flag and delay the transfer until the program ends. The last example in this chapter, MacLister, uses this second approach.

ICON TOOLS

Icons, as you know, are small symbols that usually represent disk files in the Finder. One of the first things every Macintosh owner learns is how to click and drag icons around to rearrange files in windows, copy them from disk to disk, and run programs. You can add icons to your own programs, too, or use them for any purpose you can devise.

Unfortunately, the toolbox contains only a few routines for manipulating icon images. The tools in this section correct that deficiency and demonstrate how to add icons to windows, select them, and drag them from place to place. Type in Listing 7.4 and save as ICONUNIT.PAS. Compile with Turbo to a disk code file. After the play-by-play description is an example program that explains how to use the unit.

Listing 7.4. ICONUNIT.PAS

```

1: {$O Programs:Units.F: }      { Send compiled code to here }
2: {$U-}                        { Turn off standard library units }
3:
4:
5: UNIT IconUnit( 134 );
6:
7: (*
8:
9:  * PURPOSE : Routines to display, select, and drag icons
10:  * SYSTEM  : Macintosh / Turbo Pascal
11:  * AUTHOR   : Tom Swan
12:
13: *)
14:
15:
16: INTERFACE                    { Items visible to a host program }
17:
18:
19:     USES
20:
21:         Memtypes, QuickDraw, OSIntf, ToolIntf, PackIntf;
22:
23:
24:     TYPE
25:
26:         { The following data types define a structure (and a means of
27:           accessing that structure) as it exists in an ICN# resource. }
28:
29:         IconListHandle      = ^IconListPointer;
30:         IconListPointer     = ^IconListRecord;
31:         IconListRecord      =
32:             RECORD
33:                 icon : PACKED ARRAY[ 0 .. 31 ] OF LONGINT; { Icon image }
34:                 mask : PACKED ARRAY[ 0 .. 31 ] OF LONGINT  { Mask image }
35:             END; { IconListRecord }
36:

```

```

37:
38:   { This record defines one icon image along with its location. It
39:     points to an icon list resource and keeps in destRect the icon's
40:     position in a window. }
41:
42:   IconRecord =
43:     RECORD
44:       iHand      : IconListHandle;    { Handle to icon structure }
45:       destRect   : Rect;              { Enclosing rectangle }
46:       selected   : BOOLEAN            { TRUE if icon selected }
47:     END; { IconRecord }
48:
49:
50: FUNCTION InitNewIcon(      iconID      : INTEGER;
51:                       iconRect   : Rect;
52:                       VAR anIcon   : IconRecord ) : BOOLEAN;
53:
54: PROCEDURE ShowIconImage( anIcon : IconRecord; transferMode : INTEGER );
55:
56: PROCEDURE ShowIconMask( anIcon : IconRecord; transferMode : INTEGER );
57:
58: FUNCTION IconDragged(      anIcon      : IconRecord;
59:                          mouseLoc   : Point;
60:                          wantsSlop  : BOOLEAN;
61:                          VAR newRect : Rect      ) : BOOLEAN;
62:
63: PROCEDURE SelectIcon( VAR anIcon : IconRecord );
64:
65: PROCEDURE DeSelectIcon( VAR anIcon : IconRecord );
66:
67: PROCEDURE DrawIcon( VAR anIcon : IconRecord );
68:
69: PROCEDURE MoveIcon( VAR anIcon : IconRecord; newRect : Rect );
70:
71:
72:
73:
74: IMPLEMENTATION      { Items not visible to a host program }
75:
76:
77:
78:
79: FUNCTION PositionChanged( dh, dv : INTEGER ) : BOOLEAN;
80:
81: { TRUE if dh or dv are substantial and not equal to an illegal value. }
82:
83: { LOCAL TO UNIT }
84:
85:   CONST
86:
87:     AllowableJiggle = 3;    { You can jiggle the mouse this much before}
88:                           { it's recognized as a position change. }
89:
90:     IllegalValue = $8000;  { Indicates dh and dv are not legal. }
91:
92:   BEGIN
93:
94:     PositionChanged :=
95:
96:       ( dh <> IllegalValue                                ) AND
97:
98:       ( ( ABS( dh ) > AllowableJiggle ) OR
99:         ( ABS( dv ) > AllowableJiggle ) ) )
100:
101:   END; { PositionChanged }
102:
103:

```

(continued)

```

104: FUNCTION InitNewIcon;
105:
106: { TRUE if icon with resource ID IconID is loaded and initialized }
107:
108: BEGIN
109:     WITH anIcon DO
110:     BEGIN
111:         iHand := IconListHandle( GetResource( 'ICN#', IconID ) );
112:         IF iHand = NIL
113:         THEN
114:             InitNewIcon := FALSE
115:         ELSE
116:             BEGIN
117:                 InitNewIcon := TRUE;
118:                 destRect := iconRect;
119:                 Selected := FALSE
120:             END { else }
121:         END { with }
122:     END; { InitNewIcon }
123:
124:
125: PROCEDURE ShowIconImage;
126:
127: { Display this icon's image using transferMode to copy its bits to
128:   the current grafPort. }
129:
130: VAR
131:
132:     iBitMap : BitMap;    { Icon bit map }
133:
134: BEGIN
135:     WITH anIcon, iBitMap DO
136:     BEGIN
137:         baseAddr := @iHand^.icon;          { Set pointer to icon image }
138:         rowbytes := 4;                      { Bytes in row = 32 bits }
139:         SetRect( bounds, 0, 0, 32, 32 );    { Enclosing rect }
140:
141:         CopyBits(                          { Copy image to grafPort }
142:             iBitMap,
143:             thePort^.portBits,
144:             bounds,
145:             destRect,
146:             transferMode,
147:             NIL )
148:
149:     END { with }
150: END; { ShowIconImage }
151:
152:
153: PROCEDURE ShowIconMask;
154:
155: { Display this icon's mask using transferMode to copy its bits to
156:   the current grafPort. }
157:
158: VAR
159:
160:     mBitMap : BitMap;    { Icon mask bit map }
161:
162: BEGIN
163:     WITH anIcon, mBitMap DO
164:     BEGIN
165:         baseAddr := @iHand^.mask;          { Set pointer to mask image }
166:         rowbytes := 4;                      { Bytes in row = 32 bits }
167:         SetRect( bounds, 0, 0, 32, 32 );    { Enclosing rect }
168:

```

```

169:         CopyBits(                                { Copy image to grafPort }
170:                 mBitMap,
171:                 thePort^.portBits,
172:                 bounds,
173:                 destRect,
174:                 transferMode,
175:                 NIL )
176:
177:     END { with }
178: END; { ShowIconMask }
179:
180:
181: FUNCTION IconDragged;
182:
183: { Drag an outline of this icon around the display, starting at mouseLoc.
184:   If mouse is released inside the boundaries of the current grafPort,
185:   set newRect to where the icon should be moved and return TRUE. Else
186:   return FALSE, in which case newRect is meaningless. This routine does
187:   not move the icon or change its display image in any way. Normally,
188:   before calling IconDragged, reverse the icon's image to indicate
189:   that it's selected although this is not a requirement. }
190:
191:   VAR
192:
193:       rgn          : RgnHandle;    { Region around which outline appears }
194:       limitRect    : Rect;         { Final location stays inside this rect }
195:       slopRect     : Rect;         { Mouse allowed to slop over into rect }
196:       axis         : INTEGER;      { Value to restrict horiz/vert movement }
197:       result       : LONGINT;      { Result of dragging }
198:       dh, dv       : INTEGER;      { Offsets to the new (h,v) position }
199:
200:       tlResult     : Point;        { Top Left and Bottom Right results }
201:       brResult     : Point;        { calculate limitRect to keep }
202:                                   { icon image inside window borders. }
203:
204: BEGIN
205:
206:     WITH anIcon DO
207:     BEGIN
208:
209:         rgn := NewRgn;
210:         RectRgn( rgn, destRect ); { Region for drawing dotted outline }
211:
212:         IF wantsSlop THEN
213:         BEGIN
214:             slopRect := ScreenBits.bounds; { Use screen boundaries }
215:             GlobalToLocal( slopRect.topLeft ); { Convert to local }
216:             GlobalToLocal( slopRect.botRight ) { window coordinates }
217:         END ELSE
218:             slopRect := thePort^.portRect; { No slop allowed }
219:
220:         { Set tlResult to mouseLoc - destRect.topLeft }
221:
222:         tlResult := destRect.topLeft; { Calculate difference from }
223:         SubPt( mouseLoc, tlResult ); { icon top left to mouse }
224:
225:
226:         { Set brResult to destRect.botRight - mouseLoc }
227:
228:         brResult := mouseLoc; { Calculate difference from }
229:         SubPt( destRect.botRight, brResult ); { icon bot right to mouse }
230:
231:
232:         limitRect := thePort^.portRect; { Limit final location }
233:         WITH limitRect DO
234:         BEGIN
235:             SubPt( tlResult, topLeft ); { topLeft := topLeft-tlResult }
236:             AddPt( brResult, botRight ) { botRight := botRight+brResult }
237:         END; { with }

```

(continued)

```

238:
239:
240:         axis           := 0;      { Allow movement in all directions }
241:
242:
243:     { Drag the outline, passing initialized variables to DragGrayRgn.
244:       The result is the offset to the new location. }
245:
246:     result :=
247:         DragGrayRgn( rgn, mouseLoc, limitRect, slopRect, axis, NIL );
248:
249:     dh := LoWord( result );      { Extract horizontal offset }
250:     dv := HiWord( result );      { Extract vertical offset }
251:     IF PositionChanged( dh, dv )
252:     THEN
253:         BEGIN                  { Calculate new position }
254:             newRect := destRect;
255:             OffsetRect( newRect, dh, dv );
256:             IconDragged := TRUE
257:         END { if }
258:     ELSE
259:         IconDragged := FALSE;
260:
261:     DisposeRgn( rgn )      { Dispose outline region }
262:
263:     END { with }
264:
265: END; { IconDragged }
266:
267:
268: PROCEDURE SelectIcon;
269:
270: { Select icon image, drawing it in reversed white on black. }
271:
272: BEGIN
273:     ShowIconMask( anIcon, SrcOr );      { Punch out a black mask }
274:     ShowIconImage( anIcon, SrcXor );    { Display image in white }
275:     anIcon.selected := TRUE             { Set the selected flag }
276: END; { SelectIcon }
277:
278:
279: PROCEDURE DeSelectIcon;
280:
281: { Deselect icon image, drawing it in normal black on white }
282:
283: BEGIN
284:     ShowIconMask( anIcon, SrcBic );    { Punch out a white mask }
285:     ShowIconImage( anIcon, SrcXor );    { Display image in black }
286:     anIcon.selected := FALSE           { Reset the selected flag }
287: END; { DeSelectIcon }
288:
289:
290: PROCEDURE DrawIcon;
291:
292: { Draw icon image in the current grafPort. This procedure just calls
293:   SelectIcon or DeSelectIcon, which do the actual drawing. }
294:
295: BEGIN
296:     IF anIcon.selected
297:     THEN SelectIcon( anIcon )
298:     ELSE DeSelectIcon( anIcon )
299:     END; { DrawIcon }
300:
301:
302: PROCEDURE MoveIcon;
303:

```

```

304: { Move icon from its present position to this new location.  Erases old
305:   icon by filling it with the grafPort's background pattern. }
306:
307:   BEGIN
308:     WITH anIcon DO
309:       BEGIN
310:         EraseRect( destRect );      { Erase the old image }
311:         destRect := newRect;        { Assign the new position }
312:         InvalRect( destRect )      { Force update of the new location }
313:       END { with }
314:     END; { MoveIcon }
315:
316:
317:
318: END. { IconUnit }
319:

```

IconUnit Play-by-Play

To design an icon image, type its pattern into a resource text file. For an example, look ahead to Listing 7.5 at lines 77–144 (see page 354). The hexadecimal values represent the bit patterns for an icon image, here the familiar symbol for MacPaint data files. Returning to Listing 7.4, lines 29–35 declare Pascal data types that let programs reference these hexadecimal resource structures as they exist in memory.

A single icon has two parts, an image and a mask. The image is the bit pattern you normally see. The mask is an overlay, usually restricted to the icon's borders, that programs use to reverse the normal image when you select the icon by clicking the mouse inside its symbol. Physically, each of these two parts is the same—an array of 32 long-integer values, making a two-dimensional array of 32 bits on each side. Together, one icon image and mask occupy 256 bytes in memory.

When you load an icon image from a resource file into memory, the toolbox places it in a relocatable memory block and passes a handle to your program. Because IconUnit uses this actual resource and not a copy, icon images must never be purgeable.

For keeping track of icons inside windows, IconUnit declares another record type, **IconRecord** (42–47). Field **iHand** (44) is an icon handle, locating one **IconListRecord** in memory. Rectangle **destRect** (45) is the enclosing rectangle in local coordinates that specify the icon's position and size. The final field, **selected** (46), is **TRUE** after you click the mouse inside this icon to select it or **FALSE** after you click somewhere else.

PositionChanged (79–101)

Local function **PositionedChanged** indicates whether someone has clicked and dragged an icon an appreciable distance. You cannot call this function from your own programs, but you might want to modify the way it works. As programmed here, it returns **TRUE** only if an icon's position changes by at least 4 pixels in one

direction. This lets you click an icon to select it but not change its location. If you've ever clicked a Finder icon and accidentally moved it a few pixels, you know the problem **PositionedChanged** solves. Adjust constant **AllowableJiggle** (87) to the number of pixels the mouse can travel without moving the icon.

```
FUNCTION InitNewIcon( iconID : INTEGER;
  iconRect : Rect; VAR anIcon : IconRecord ) :
  BOOLEAN;
```

Pass an icon resource ID number in **iconID**, an enclosing 32×32 rectangle in **iconRect**, and an icon record in **anIcon** to **InitNewIcon** (104–122) for each icon that your program uses. If it returns **TRUE**, then it was able to load the icon image into memory and initialize the fields in **anIcon**.

The function calls **GetResource** (111) to load the icon list resource (ICN#) from disk. It casts the resulting handle to type **IconListHandle**, assigning it to field **iHand** in the icon record. If this value is **NIL**, indicating that **GetResource** could not load the icon, the function returns **FALSE** (114). Otherwise, it continues at lines 117–119 to initialize the remaining icon record fields and return **TRUE**.

```
PROCEDURE ShowIconImage( anIcon : IconRecord;
  transferMode : INTEGER );
```

After loading an icon list resource with **InitNewIcon**, to display the image, pass the resulting icon record to **ShowIconImage** (125–150). Set parameter **transferMode** to one of the following constants:

SrcCopy	NotSrcCopy	SrcOr	NotSrcOr
SrcXor	NotSrcXor	SrcBic	NotSrcBic

These eight modes change the way QuickDraw procedure **CopyBits** transfers bit images from one place to another, usually the display. (See Chapter 3 for more details about this subject.) **ShowIconImage** assigns the address of your icon image to a local **BitMap** record (137), setting fields **rowBytes** and **bounds** to enclose the image in memory (138–139). It then passes this information to **CopyBits** (141–147), transferring the icon's bit image to the current **GrafPort** and displaying its pattern on screen. The **NIL** parameter (147) tells QuickDraw not to clip the image in any way.

```
PROCEDURE ShowIconMask( anIcon : IconRecord;
  transferMode : INTEGER );
```

ShowIconMask (153–178) is nearly identical to **ShowIconImage**. It differs by displaying the icon mask instead of the image. To display a single icon, you first display the mask, “punching” a hole in the screen like a cookie cutter punching

out dough. You then display the icon image by dropping it into the hole created by the mask. By doing this, instead of simply displaying the image in one step, you create transparent icons whose background patterns match their masks. This lets you highlight images for various effects. (For an example, see **SelectIcon**.)

```
FUNCTION IconDragged( anIcon : IconRecord;  
    mouseLoc : Point; wantsSlop : BOOLEAN;  
    VAR newRect : Rect ) : BOOLEAN;
```

Call **IconDragged** (181–265) after sensing a mouse click inside an icon. The routine pulls an outline of the icon around the display, releasing control only when you let go of the mouse button. Parameter **anIcon** is the icon's record, initialized as described earlier. **Point** record **mouseLoc** is the mouse pointer's coordinate which you obtain by calling **GetMouse**. The coordinate values are local to the current window.

If **wantsSlop** is **TRUE**, you can move the mouse pointer outside the current window while restricting the final icon location to inside the window's borders. Set this parameter **FALSE** if you want to cancel icon dragging when the mouse pointer moves outside. If you then release the mouse button, the icon's original position is unchanged. (The next program demonstrates the difference between these two options.)

If **IconDragged** returns **TRUE**, then rectangle **newRect** equals the location to which you should move the icon image. Normally, you'll do this by calling **MoveIcon**. See Listing 7.6, lines 54–76 for a complete example.

Although long, the procedure is not difficult to understand. To draw the dotted outline requires a region, which **IconDragged** creates at lines 209–210, using the **destRect** field from the icon record. Lines 212–218 set a *slop* rectangle either to the entire screen boundaries in local coordinates (214–216) or to the window's border (218) depending on whether you set **wantsSlop** to **TRUE** or **FALSE**. As long as the mouse pointer remains inside this rectangle, the dotted outline is visible.

Lines 222–237 set up another rectangle, **limitRect**, restricting the final icon location to within the window's boundaries. The calls to **SubPt** (Subtract Points) calculate this rectangle to fall within the window borders while taking into account the mouse location, which might be anywhere inside the icon. This lets you move the icon to every screen location regardless of where you click on the image. If this is unclear, change lines 233–237 to a comment, setting **limitRect** to the full window border. Then run the icon test program in Listing 7.6. You can now force the icon outside the window borders, probably a poor idea although it does no harm.

Line 240 assigns zero to variable **axis**, allowing movement in any direction. Set this value to toolbox constant **hAxisOnly** for horizontal movement only, or set it to **vAxisOnly** to allow only vertical moves. If you need to do this often, you might want to add **axis** to the procedure's parameter list instead of declaring it as a local variable as it is now.

The actual dragging occurs with a call to the Window Manager's **DragGrayRgn**

routine, which lets you pull the dotted outline around the display (246–247). The final **NIL** parameter indicates no action procedure, an option that you could use to call another routine while you hold down the mouse button. (See *Inside Macintosh* for details about how to do this.)

DragGrayRgn returns a single long integer result containing offsets to the new icon position. Lines 249–259 extract the components of this result, assigning the horizontal and vertical offsets to **dh** and **dv**. The **IF** statement (251) checks whether these values are large enough to require moving the icon image and, if so, lines 254–256 pass the new rectangle back in parameter **newRect**, telling the caller the new location.

The final statement (261) is an easy one to forget. It disposes the region handle created at the start of this procedure. Remember, as repeated often in this book, always dispose your handles when you're finished using them.

```
PROCEDURE SelectIcon( VAR anIcon : IconRecord );
PROCEDURE DeSelectIcon( VAR anIcon :
    IconRecord );
```

These two procedures complement each other. **SelectIcon** (268–276) darkens the icon image, presumably to respond to mouse clicks. **DeSelectIcon** (279–287) does the opposite, redrawing a highlighted image normally.

Both routines set field **selected** to **TRUE** or **FALSE**. You can use this field to know whether an icon is selected (highlighted). To manage many icons at once, you could put their records in an array and check for selected patterns by examining **selected** fields in each one. Selecting an icon is a visual effect only and causes no other actions to occur. It's up to you to decide what to do with selected icons.

Line 273 punches out a black mask in the window, using transfer mode **SrcOr**. Because a logical OR sets bits to 1 if either or both of two bits are 1, the result is to blacken the screen for every bit in the mask, usually covering the entire icon image. After that, line 274 calls **ShowIconImage** to drop in the icon image inside the blackened mask. Because it uses the **SrcXor** transfer mode, the effect is to negate bits only where they differ, which displays the icon image in white on the black background.

DeSelectIcon (279–287) uses a similar method to re-reverse the highlighted icon, displaying it normally. Transfer mode **SrcBic** (Source Bit Clear) also punches a mask on the screen but this time in white (284). Line 285 then uses the same **SrcXor** mode to drop the icon image into this area, displaying its pattern as black lines on a white background.

It's instructive to turn a few of lines 273–274 and 284–285 into comments in various combinations. (If you comment out all of these lines, though, you'll see nothing on screen.) Also, try other transfer modes listed earlier in this chapter and in Chapter 3. You can create a variety of effects by simply calling tools **SelectIconMask** and **SelectIconImage** with various transfer mode values.

```
PROCEDURE DrawIcon( VAR anIcon : IconRecord );
```

DrawIcon (290–299) simplifies your program's update event handler. Call this routine for every icon your program uses, passing their records in parameter **anIcon**. The procedure draws each icon by calling **SelectIcon** or **DeSelectIcon** according to the value of field **selected**.

```
PROCEDURE MoveIcon( VAR anIcon : IconRecord;
  newRect : Rect );
```

Use **MoveIcon** (302–314) to move an icon image from its present location to a new spot with coordinates in parameter **newRect**. The procedure first erases the icon image (310) and then assigns the new rectangle to the icon record's **destRect** field. Rather than copying the image directly into the display at this point, **MoveIcon** invalidates the rectangle (312), adding its area to the window's update region and generating an update event. The actual drawing takes place in your program's event handler (which probably calls **DrawIcon**).

CLICKING AND DRAGGING ICONS

For an example of how to use IconUnit, type in Listing 7.5 and save as ICONTEST.R. Compile this file with RMaker to produce the program's resource file. Then type in Listing 7.6 and save as ICONTEST.PAS. Copy lines from Ap-Shell where the listing indicates. Compile and run with Turbo. Inside the window, you'll see a single icon, which you can click and drag anywhere inside the window borders. After trying the program, experiment with the suggestions in the previous section. For example, change the **axis** variable (Listing 7.4, 240) to limit movement in one direction or another.

Listing 7.5. ICONTEST.R

```
1: *-----*
2: * IconTest.PAS resources -- Compile with RMaker *
3: *-----*
4:
5: Programs:Icon.F:IconTest.RSRC    ;; Send output to here
6:
7:
8: *-----*
9: * About box string list *
10: *-----*
11:
12: TYPE STR#                      ;; String list resource
13:     ,1 (32)                    ;; ID and attribute (purgeable)
14: 6                               ;; Number of strings that follow
15: Icon Test                      ;; Program name
16: by Tom Swan                   ;; Author
```

(continued)

```

17: Version 1.00                ;; Version number
18: (C) 1987 by Swan Software   ;; Copyright notice
19: P. O. Box 206, Lititz, PA 17543 ;; Address
20: (717)-627-1911             ;; Telephone
21:
22:
23: *-----*
24: * The Apple Info menu      *
25: *-----*
26:
27: TYPE MENU
28:     ,1
29: \14
30:     About IconTest...
31:     (-
32:
33:
34: *-----*
35: * The File menu            *
36: *-----*
37:
38: TYPE MENU
39:     ,2
40: File
41:     Quit /Q
42:
43:
44: *-----*
45: * The Edit menu            *
46: *-----*
47:
48: TYPE MENU
49:     ,3
50: Edit
51:     (Undo /Z
52:     (-
53:     (Cut /X
54:     (Copy /C
55:     (Paste /V
56:     (Clear
57:
58:
59: *-----*
60: * Window template          *
61: *-----*
62:
63: TYPE WIND
64:     ,1                ;; ID number to use in program
65: IconUnit Test         ;; Window title
66: 46 7 328 502          ;; top, left, bottom, right coordinates
67: Visible NoGoAway      ;; Visible window without close button
68: 4                     ;; Document window style without size button
69: 0                     ;; Window reference (none)
70:
71:
72: *-----*
73: * MacPaint Icon list      *
74: *-----*
75:
76:
77: Type ICN# = GNRL      ;; General type (its structure is up to you)
78:     ,1000             ;; Resource ID
79: .H                   ;; Hex data follows
80: OFFFFE00             ;; 32 LONGINTS (32-bits each)
81: 08000300
82: 09D00280
83: 09D00240

```

```

84: 09D00220
85: 09D00210
86: 09D003F8
87: 09D00008
88: 09D00008
89: 09D00008
90: 09D00008
91: 09D00008
92: 09F00008
93: 09100008
94: 09100008
95: 09100008
96: 09100008
97: 09100008
98: 08E00008
99: 09F00008
100: 09F00008
101: 09F80008
102: 09F80008
103: 09E85FE8
104: 09F80BE8
105: 08D03FE8
106: 08F0FFE8
107: 08703FE8
108: 0819FFE8
109: 08000008
110: 08000008
111: 0FFFFFFF8
112: *
113: 0FFFFFFE0
114: 0FFFFFF00
115: 0FFFFFF80
116: 0FFFFFFC0
117: 0FFFFFFE0
118: 0FFFFFF00
119: 0FFFFFFF8
120: 0FFFFFFF8
121: 0FFFFFFF8
122: 0FFFFFFF8
123: 0FFFFFFF8
124: 0FFFFFFF8
125: 0FFFFFFF8
126: 0FFFFFFF8
127: 0FFFFFFF8
128: 0FFFFFFF8
129: 0FFFFFFF8
130: 0FFFFFFF8
131: 0FFFFFFF8
132: 0FFFFFFF8
133: 0FFFFFFF8
134: 0FFFFFFF8
135: 0FFFFFFF8
136: 0FFFFFFF8
137: 0FFFFFFF8
138: 0FFFFFFF8
139: 0FFFFFFF8
140: 0FFFFFFF8
141: 0FFFFFFF8
142: 0FFFFFFF8
143: 0FFFFFFF8
144: 0FFFFFFF8
145:
146:
147:
148: * END

```

Mask data

Listing 7.6. ICONTEST.PAS

```

1: {$O Programs:Icon.F: }           { Send compiled code to here }
2: {$R Programs:Icon.F:IconTest.Rsrc} { Use this compiled resource file }
3: {$U-}                             { Turn off standard library units }
4:
5:
6: PROGRAM IconTest;
7:
8: (*
9:
10:  * PURPOSE : Test the IconUnit -- display, select, and drag an icon
11:  * SYSTEM  : Macintosh / Turbo Pascal
12:  * AUTHOR  : Tom Swan
13:
14: *)
15:
16:
17: {$U Programs:Units.F:MacExtras }   { Open these library unit files }
18: {$U Programs:Units.F:IconUnit }
19:
20:
21:   USES
22:
23:       Memtypes, QuickDraw, OSIntf, ToolIntf, PackIntf, MacExtras,
24:       IconUnit;
25:
26:
27:   CONST
28:
29:       FileID      = 2;      { File menu Resource ID and commands }
30:       QuitCmd     = 1;
31:
32:       WindowID    = 1;      { Window resource ID }
33:
34:       IconID      = 1000;   { Icon list resource ID }
35:
36:       WantsSlop   = TRUE;   { FALSE for no mouse slop outside window }
37:
38:       top         = 10;     { Initial position of icon in window }
39:       left        = 10;
40:
41:
42:
43:   VAR
44:
45:       wRec        : WindowRecord;      { Program's window data record }
46:       wPtr        : WindowPtr;         { Pointer to above wRec }
47:
48:       quitRequested : BOOLEAN;          { TRUE if quitting }
49:
50:       theIcon      : IconRecord;        { Icon image and information }
51:
52:
53:
54: PROCEDURE DoMouseClicked;
55:
56: { Handle mouse clicks in wPtr's contents. Select icon and drag outline
57:  around screen, dropping icon off in its new position. }
58:
59:   VAR
60:
61:       mouseLoc : Point;
62:       newRect  : Rect;
63:

```

```

64: BEGIN
65:     GetMouse( mouseLoc );
66:     WITH theIcon DO
67:         IF NOT PtInRect( mouseLoc, destRect ) { Mouse outside icon }
68:         THEN
69:             DeSelectIcon( theIcon )
70:         ELSE
71:             BEGIN
72:                 SelectIcon( theIcon );
73:                 IF IconDragged( theIcon, mouseLoc, WantsSlop, newRect )
74:                 THEN MoveIcon( theIcon, newRect )
75:             END { else }
76:         END; { DoMouseClicked }
77:
78:
79: PROCEDURE DoFileMenuCommands( cmdNumber : INTEGER );
80:
81: { Execute command in the File menu }
82:
83: BEGIN
84:     quitRequested := ( cmdNumber = QuitCmd )
85: END; { DoFileMenuCommands }
86:
87:
88: << INSERT LINES 137-175 FROM APSHELL.PAS >>
89:
90:
91: PROCEDURE DrawContents( whichWindow : WindowPtr );
92:
93: { Display window contents }
94:
95: BEGIN
96:
97:     DrawIcon( theIcon )
98:
99: END; { DrawContents }
100:
101:
102: PROCEDURE MouseDownEvents;
103:
104: { Someone pressed the mouse button. Check its location and respond. }
105:
106: VAR
107:
108:     partCode : INTEGER;      { Identifies what item was clicked. }
109:
110: BEGIN
111:
112:     WITH theEvent DO
113:
114:         BEGIN
115:
116:             partCode := FindWindow( where, whichWindow );
117:
118:             CASE partCode OF
119:
120:                 inMenuBar
121:                     : DoCommand( MenuSelect( where ) );
122:
123:                 inSysWindow
124:                     : SystemClick( theEvent, whichWindow );
125:
126:                 inContent, inDrag
127:                     : IF whichWindow <> FrontWindow
128:                       THEN SelectWindow( whichWindow )
129:                       ELSE IF partCode = inContent
130:                           THEN DoMouseClicked
131:
132:             END { case }

```

(continued)

```

133:
134:         END { with }
135:
136:     END; { MouseDownEvents }
137:
138:
139: PROCEDURE KeyDownEvents;
140:
141: { A key was pressed. Do something with incoming character. }
142:
143:     VAR
144:
145:         ch : CHAR;
146:
147:     BEGIN
148:         WITH theEvent DO
149:             BEGIN
150:                 ch := CHR( BitAnd( message, charCodeMask ) );
151:                 IF BitAnd( modifiers, CmdKey ) <> 0
152:                     THEN DoCommand( MenuKey( ch ) )
153:                 END { with }
154:             END; { KeyDownEvents }
155:
156:
157: << INSERT LINES 301-318 FROM APSHELL.PAS >>
158:
159:
160: PROCEDURE ActivateEvents;
161:
162: { Activate or deactivate windows }
163:
164:     BEGIN
165:         WITH theEvent DO
166:             BEGIN
167:
168:                 whichWindow := WindowPtr( message ); { Extract window pointer }
169:                 SetPort( whichWindow ); { Change current port }
170:
171:                 IF BitAnd( modifiers, activeFlag ) <> 0
172:                     THEN FixEditMenu( FALSE ) { Activate a window }
173:                     ELSE FixEditMenu( TRUE ) { Deactivate a window }
174:
175:                 END { with }
176:             END; { ActivateEvents }
177:
178:
179: << INSERT LINES 350-368 FROM APSHELL.PAS >>
180:
181:
182: PROCEDURE SetUpWindow;
183:
184: { Initialize this program's window record }
185:
186:     BEGIN
187:         wPtr := GetNewWindow( WindowID, @Wrec, POINTER(-1) );
188:         SetPort( wPtr )
189:     END; { SetUpWindow }
190:
191:
192: PROCEDURE SetUpIcon;
193:
194: { Initialize icon variable, loading image from resource file.
195:   Assumes wPtr is the current port }
196:

```

```

197:  VAR
198:
199:      iconRect : Rect;
200:
201:  BEGIN
202:      SetRect( iconRect, left, top, left + 32, top + 32 );
203:      IF NOT InitNewIcon( IconID, iconRect, theIcon ) THEN
204:          BEGIN
205:              SysBeep(3);      { If you hear a beep and the program }
206:              ExitToShell      { ends, check your resource definition. }
207:          END { if }
208:      END; { SetUpIcon }
209:
210:
211:  PROCEDURE Initialize;
212:
213:  { Program calls this routine one time at start }
214:
215:  BEGIN
216:      SetUpMenuBar;           { Initialize and display menus }
217:      SetUpWindow;           { Initialize program window }
218:      SetUpIcon;             { Initialize icon image }
219:      quitRequested := FALSE; { TRUE on selecting Quit command }
220:      DisplayAboutBox        { Identify program }
221:  END; { Initialize }
222:
223:
224:  FUNCTION QuitConfirmed : BOOLEAN;
225:
226:  { The program's "deinitialization" routine. It is always okay
227:  to quit this program. }
228:
229:  BEGIN
230:      QuitConfirmed := quitRequested
231:  END; { QuitConfirmed }
232:
233:
234:  PROCEDURE DoSystemTasks;
235:
236:  { Do operations at each pass through main program loop }
237:
238:  BEGIN
239:
240:      SystemTask;      { Give DAS their fair share of time }
241:
242:      IF FrontWindow = NIL THEN
243:
244:          BEGIN { Set up menu commands for empty desktop }
245:
246:              FixEditMenu( FALSE )
247:
248:          END ELSE
249:
250:          IF FrontWindow <> wPtr THEN
251:
252:              BEGIN { Set up menu commands for active desk accessory }
253:
254:                  FixEditMenu( TRUE )
255:
256:              END { else / if }
257:
258:          END; { DoSystemTasks }
259:
260:

```

(continued)


```

261: BEGIN
262:
263:     Initialize;
264:
265:     REPEAT
266:
267:         DoSystemTasks;
268:
269:         IF GetNextEvent( everyEvent, theEvent ) THEN
270:
271:             CASE theEvent.what OF
272:
273:                 MouseDown    : MouseDownEvents;
274:                 KeyDown       : KeyDownEvents;
275:                 UpdateEvt      : UpdateEvents;
276:                 ActivateEvt    : ActivateEvents
277:
278:             END { case }
279:
280:         UNTIL QuitConfirmed
281:
282:     END.

```

IconTest Play-by-Play

ICONTEST.R (1-148)

Most of the resource types are the same as in ApShell. To simplify the test program, the window resource is immovable and does not have close and zoom boxes. The only new element is the icon list resource (77-144).

Type ICN# (the # stands for “list”) is a GNRL (general) structure, meaning its design is up to you (see line 77). As in other resources, the resource ID is a positive number, also of your choice, but probably best if at least 128 or higher. This number matches the ID the program uses to load the resource into memory. The test uses ID 1000 (78). Remember that icon lists must never be purgeable.

The **.H** (79) signifies that hex data follows. Here, that data consists of two blocks of 32, 32-bit values, each having 8 hex digits (worth four bits apiece). Each bit in the data represents one black pixel in the icon image. The first block of 32 values (80-111) is the image; the second (113-144) is the mask that overlays the image. Remember that this mask should completely cover the icon outline, or highlighting might not work properly. It is possible to design masks with holes, though, to animate icons, changing their patterns when you select them, a popular technique. IconTest is a good program for experimenting with such effects. Just replace lines 80-144 with a new image and mask.

ICONTEST.PAS (1-176)

Constant **IconID** (34) matches the ID number of the icon list resource. In a program with several icons, you would define an ICN# resource for each one and assign them unique IDs. Set Boolean constant **WantsSlop** (36) **TRUE** to allow slop-

py mouse movement, or to **FALSE** to erase the dragging outline if the mouse pointer strays outside the window borders. Experiment with both values to understand their difference. Two other constants, **Top** and **Left** (38–39) mark the upper left corner of the icon's initial position. They can be any values, relative to the window coordinates.

Procedure **DoMouseClicked** (54–76) responds to mouse clicks inside windows that contain icons. The first job is to locate the mouse, calling **GetMouse** (65), which fills in the **mouseLoc Point** record with a coordinate relative (local) to the window borders. Function **PtInRect** then tests whether that point is inside the icon's enclosing rectangle (67). If not, **DeSelectIcon** redraws the icon normally, setting icon record field **selected** to **FALSE** (69). (Deselecting an icon not previously selected has no effect.) If the mouse click is inside the image, lines 72–74 select the icon and call **IconDragged** pulling a dotted outline around the screen. If **IconDragged** returns **TRUE**, **MoveIcon** (74) completes the process by moving the image to its new location in the window.

Procedure **DrawContents** (91–99) is this program's entire display handler. It calls **DrawIcon** to draw the icon image, highlighting it only if field **selected** in the icon record is **TRUE**. To handle many icons, perhaps in an array of icon records, call **DrawIcon** in this routine for each one.

Other procedures in this section are shortened versions of those with the same names in ApShell. You should have little trouble understanding what they do.

SetUpWindow to END (182–282)

The test program's only window appears as a result of **SetUpWindow** (182–189), which loads the window resource and sets the current **GrafPort** to the window pointer. Of course you can use **IconUnit** with other window types: those with resize boxes, scroll bars, and other features.

The next procedure, **SetUpIcon** (192–208), initializes new icon images. In line 202, it creates a 32×32 pixel rectangle (**iconRect**), which it passes along with the icon resource ID (**IconID**) and an icon record variable (**theIcon**) to **InitNewIcon** (203). This loads the **ICN#** resource into memory and initializes the icon record's fields. In this example, if that doesn't work, then something is wrong with the resource file and the program beeps before returning to Turbo, or to the Finder if you run it from a disk code file. Therefore, if you hear a beep and nothing else seems to happen, check your **ICN#** resource against Listing 7.5.

PRINTING TOOLS

The Macintosh toolbox contains many routines for printing graphics and text. As you probably know, you install printer drivers in your System folder to use different printers without requiring programs to understand their specific characteristics. To programs, all printers are the same. It's the software in the driver

that takes advantage of special abilities such as the fonts in a laser printer or the graphics in an ImageWriter.

To print a picture or text in this fashion, programs draw into memory (or into temporary disk files) similar to the way they draw onto the display. But instead of that drawing appearing on screen, it appears on paper—the result of the interaction between the operating system and the printer driver. Because of that, programs don't have to know which commands turn on graphics or underline text for one printer or another.

Unfortunately, this ideal arrangement takes time—time to draw into memory, time to feed that drawing to the printer driver, and time for the driver to print the dots on paper that create the final image. Consequently, the printer runs more slowly than it does when it uses its own native abilities to print text.

Because the standard printer methods are well covered elsewhere, I won't repeat them here. Instead, the next section develops a set of tools to use an ImageWriter's faster native text abilities. With these tools, you select among various printing features such as condensed and proportional type and headlines—but you cannot print graphics or fonts that you see on screen. With a few simple modifications, you can use this same programming to run other printers, even the letter-quality, daisy- and thimble-wheel models that the Macintosh normally doesn't know how to operate.

One thing to remember is that these tools break all the Macintosh rules about printing. If you write a program to work with the ImageWriter and then hook up a laser printer, the program might not work properly. Despite this restriction, the following tools greatly increase printing speed, a welcome improvement for program listings, database reports, and simple notes.

To begin adding native ImageWriter tools to your system, type in Listing 7.7 and save as IMAGEUNIT.PAS. Compile the text to a disk code file. The unit does not use any other units in this book. Following the play-by-play description is an example program that uses the unit to print program listings.

Listing 7.7. IMAGEUNIT.PAS

```

1: {$O Programs:Units.F: }      { Send compiled code to here }
2: {$U-}                        { Turn off standard library units }
3:
4:
5: UNIT ImageUnit( 133 );
6:
7: (*
8:
9:  * PURPOSE : Imagewriter "native" printing routines
10:  * SYSTEM  : Macintosh / Turbo Pascal
11:  * AUTHOR   : Tom Swan
12:

```

```

13:  *)
14:
15:
16: INTERFACE      { Items visible to a host program }
17:
18:
19:     USES
20:
21:     MemTypes, QuickDraw, OSIntf, ToolIntf, PackIntf, MacPrint;
22:
23:
24:     TYPE
25:
26:         PrnStyles =
27:         ( PrnNoStyle, PrnExtended, PrnPica, PrnElite,
28:           PrnPicaPro, PrnElitePro, PrnSemiCond, PrnCondensed,
29:           PrnUltraCond, PrnHLStart, PrnHLEnd );
30:
31:         PrnHandle = ^PrnPtr;      { Handle to relocatable PrnSpecRec }
32:         PrnPtr = ^PrnSpecRec;    { Pointer to PrnSpecRec record }
33:
34:         PrnSpecRec =
35:         RECORD                  { Various printer variables as follows }
36:
37:             lut   : INTEGER;    { Blank lines under top header }
38:             laf   : INTEGER;    { Blank lines above bottom footer }
39:             cpl   : INTEGER;    { Maximum characters per line }
40:             lpp   : INTEGER;    { Total lines per page }
41:             ltp   : INTEGER;    { Lines to print on page (not headers) }
42:             lsp   : INTEGER;    { Line spacing. 0=single,1=double,etc. }
43:             tab   : INTEGER;    { Fixed tab stop. Usually 4 or 8. }
44:             spn   : INTEGER;    { Starting page number (usually 1) }
45:             sln   : INTEGER;    { Starting line number (usually 1) }
46:
47:             header : Str255;    { Header string to print at top }
48:             footer  : Str255;   { Footer string to print at bottom }
49:
50:             printHeader : BOOLEAN; { True for headers in top margin }
51:             printFooter : BOOLEAN; { True for footers in bottom margin }
52:             hasFormFeed : BOOLEAN; { True if ASCII 12 does form feed }
53:             lineNumbers : BOOLEAN; { True to print line numbers }
54:
55:             textStyle : PrnStyles; { Printer style for text }
56:
57:         END; { PrnSpecRec }
58:
59:
60:     PROCEDURE PrnSetStyle( pStyle : PrnStyles );
61:
62:     PROCEDURE PrnChar( ch : CHAR );
63:
64:     PROCEDURE PrnString( s : Str255 );
65:
66:     PROCEDURE PrnLine( s : Str255; lastLine : BOOLEAN );
67:
68:     FUNCTION PrnNew : PrnHandle;
69:
70:     PROCEDURE PrnDispose( VAR pHand : PrnHandle );
71:
72:     PROCEDURE PrnStart( pHand : PrnHandle );
73:
74:     PROCEDURE PrnEnd;
75:
76:
77: IMPLEMENTATION      { Items not visible to a host program }
78:

```

(continued)

```

79:  CONST
80:
81:      ASCTab      = #9;      { ASCII character constants }
82:      ASCCr       = #13;
83:      ASCLf       = #10;
84:      ASCff       = #12;
85:      ASCesc      = #27;
86:      ASCblank    = #32;
87:
88:      MaxLNDigits = 3;      { Maximum digits in line numbers 000-999 }
89:      MaxPNDigits = 2;      { Maximum digits in page numbers 00-99 }
90:
91:
92:  VAR
93:
94:      prnSpec : PrnHandle;    { Handle to printer specs }
95:
96:      dtpString : Str255;      { Date, time, page number string }
97:
98:      pgLine      : INTEGER;   { Line number (resets at top of page) }
99:      column      : INTEGER;   { Column number relative to left margin }
100:     lineNo       : INTEGER;   { Running line number }
101:     pageNo       : INTEGER;   { Page number }
102:     lutltp       : INTEGER;   { lut + ltp (see PrnSpecRec type) }
103:     lutltplaf    : INTEGER;   { lut + ltp + laf ( " " " " ) }
104:     titleOverHead : INTEGER;  { Length of date, time, & page no string }
105:     lastLineDone  : BOOLEAN;  { TRUE when printing the last line }
106:
107:  PROCEDURE ListChar( ch : CHAR );
108:
109:  { Send character ch to list device using text streaming method.
110:    LOCAL TO UNIT }
111:
112:  VAR
113:
114:      buffer : PACKED ARRAY[ 0 .. 0 ] OF CHAR;
115:
116:  BEGIN
117:      buffer[ 0 ] := ch; { Move ch value to high order 8 bits }
118:      PrCtlCall( iPrIOCtl, ORD( @buffer ), 1, 0 )
119:  END; { ListChar }
120:
121:
122:  PROCEDURE PrnTab;
123:
124:  { Advance printer to next fixed tab stop.
125:    LOCAL TO UNIT }
126:
127:  VAR
128:
129:      tabWidth : INTEGER;
130:
131:  BEGIN
132:      tabWidth := prnSpec^.tab; { Copy value--ListChar could compact heap }
133:      REPEAT
134:          ListChar( ASCblank );
135:          column := column + 1
136:      UNTIL column MOD tabWidth = 0
137:  END; { PrnTab }
138:
139:
140:  PROCEDURE PrnLf;
141:
142:  { Advance printer to next line down. No cr. Advance line counts.
143:    LOCAL TO UNIT }
144:

```

```

145: BEGIN
146:   ListChar( ASClf );
147:   pgLine := pgLine + 1    { Count relative line number }
148: END; { PrnLf }
149:
150:
151: PROCEDURE PrnCr;
152:
153: { Send printer to start of current line. No lf.
154:   LOCAL TO UNIT }
155:
156: BEGIN
157:   ListChar( ASCcr );
158:   column := 0
159: END; { PrnCr }
160:
161:
162: PROCEDURE PrnAdvanceLine( n : INTEGER );
163:
164: { Advance printer n lines down
165:   LOCAL TO UNIT }
166:
167: BEGIN
168:   PrnCr;
169:   WHILE n > 0 DO
170:     BEGIN
171:       PrnLf;
172:       n := n - 1
173:     END { while }
174:   END; { PrnAdvanceLine }
175:
176:
177: PROCEDURE PrnFf;
178:
179: { Advance printer to next page
180:   LOCAL TO UNIT }
181:
182: BEGIN
183:   WITH prnSpec^^ DO
184:     IF hasFormFeed
185:       THEN ListChar( ASCff )
186:     ELSE IF ( pgLine <= lpp )
187:       THEN PrnAdvanceLine( lpp - pgLine ); { Zero does lone cr }
188:     pgLine := 0 { reset relative line number }
189:   END; { PrnFf }
190:
191:
192: PROCEDURE PrnFooter;
193:
194: { Print footer at bottom of page.
195:   LOCAL TO UNIT }
196:
197: VAR
198:
199:   title : Str255;      { Temporary variable to hold footer text }
200:   blanks : INTEGER;    { Number of blank lines or chars to print }
201:
202: BEGIN
203:   blanks := luttltplaf - pgLine + 1;    { Blank lines above footer }
204:   IF blanks > 0
205:     THEN PrnAdvanceLine( blanks );      { Advance to footer line }
206:   IF prnSpec^^.printFooter THEN
207:     BEGIN
208:       PrnSetStyle( PrnHLStart );        { Turn on headline style }
209:       title := prnSpec^^.footer;        { Copy title to local var }
210:       blanks := ( prnSpec^^.cpl DIV 2 ) - length( title );
211:       WHILE blanks > 0 DO

```

(continued)

```

212:         BEGIN
213:             ListChar( ASCblank );           { Move to right border }
214:             blanks := blanks - 1
215:         END; { while }
216:         prnString( title );                 { Print title }
217:         PrnSetStyle( PrnHLEnd );            { Turn off headline style }
218:         PrnSetStyle( prnSpec^.textStyle )   { Restore text style }
219:     END; { if }
220:     PrnFf                                     { Advance to top of next page }
221: END; { PrnFooter }
222:
223:
224: PROCEDURE PrnHeader;
225:
226: { Print header at top of page.
227:  LOCAL TO UNIT }
228:
229:  VAR
230:
231:      title : Str255;
232:      blanks : INTEGER;
233:
234:  BEGIN
235:      IF prnSpec^.printHeader THEN
236:          BEGIN
237:              PrnSetStyle( PrnHLStart );      { Turn on headline style }
238:              title := prnSpec^.header;       { Copy title to local var }
239:              PrnString( title );              { Print title at left border }
240:              blanks := ( prnSpec^.cpl DIV 2 ) - length(title) - titleOverHead;
241:              WHILE blanks > 0 DO
242:                  BEGIN
243:                      ListChar( ASCblank );   { Advance to date position }
244:                      blanks := blanks - 1
245:                  END; { while }
246:                  PrnString( dtpString );      { Print date, time, page no. }
247:                  NumToString( pageNo, title ); { Reuse title string to print }
248:                  PrnString( title );          { page number. }
249:                  PrnSetStyle( PrnHLEnd );     { Turn off headline style }
250:                  PrnSetStyle( prnSpec^.textStyle ) { Restore text style }
251:              END; { if }
252:              PrnAdvanceLine( prnSpec^.lut + 1 ) { Blank lines under title }
253:          END; { PrnHeader }
254:
255:
256: PROCEDURE PrnNewPage;
257:
258: { Start new page.
259:  LOCAL TO UNIT }
260:
261:  BEGIN
262:      IF NOT lastLineDone THEN
263:          BEGIN
264:              PrnFooter;
265:              pgLine := 0;
266:              pageNo := pageNo + 1;
267:              PrnHeader
268:          END { if }
269:      END; { PrnNewPage }
270:
271:
272: PROCEDURE PrnNewLine;
273:
274: { Start new line.
275:  LOCAL TO UNIT }
276:
277:  BEGIN
278:      IF pgLine >= lutltp
279:          THEN PrnNewPage
280:          ELSE PrnAdvanceLine( prnSpec^.lsp + 1 )
281:      END; { PrnNewLine }

```

```

282:
283:
284: PROCEDURE PrnSetStyle;
285:
286: { Select one of various print styles.  Modify for different
287:   printer models. }
288:
289:   VAR
290:
291:       ch : CHAR;
292:
293:   BEGIN
294:       CASE pStyle OF
295:           { PrnNoStyle }
296:               PrnExtended      : ch := 'n';
297:               PrnPica          : ch := 'N';
298:               PrnElite         : ch := 'E';
299:               PrnPicaPro       : ch := 'p';
300:               PrnElitePro      : ch := 'P';
301:               PrnSemiCond      : ch := 'e';
302:               PrnCondensed     : ch := 'q';
303:               PrnUltraCond     : ch := 'Q';
304:               PrnHLStart       : BEGIN
305:                                   ListChar( ^N ); { Start headline mode }
306:                                   ch := '!'         { Start boldface }
307:                               END;
308:               PrnHLEnd         : BEGIN
309:                                   ListChar( ^O ); { End headline mode }
310:                                   ch := '""'       { End boldface mode }
311:                               END
312:           END; { case }
313:       ListChar( ASCesc ); { Send lead-in character }
314:       ListChar( ch )      { Send printer command character }
315:   END; { PrnSetStyle }
316:
317:
318: PROCEDURE PrnChar;
319:
320: { Print one character.  Specially handle some control chars. }
321:
322:   BEGIN
323:       IF ch = ASCtab THEN PrnTab
324:       ELSE IF ch = ASClf THEN PrnLf
325:       ELSE IF ch = ASCcr THEN PrnCr
326:       ELSE IF ch = ASCff THEN Prnff
327:       ELSE
328:           BEGIN
329:               ListChar( ch );
330:               IF ch >= ASCblank THEN column := column + 1
331:           END { else }
332:       END; { PrnChar }
333:
334:
335: PROCEDURE PrnString;
336:
337: { Print string parameter--no carriage return (similar to Write) }
338:
339:   VAR
340:
341:       i : INTEGER;
342:
343:   BEGIN
344:       FOR i := 1 TO Length( s ) DO
345:           PrnChar( s[i] )
346:       END; { PrnString }
347:
348:

```

(continued)


```

349: PROCEDURE PrnLine;
350:
351: { Print an entire line (similar to WriteLn) }
352:
353:   VAR
354:
355:       numString : Str255; { For printing line numbers }
356:
357: BEGIN
358:   lastLineDone := lastLine;
359:   IF prnSpec^.lineNumbers THEN
360:     BEGIN { print line number }
361:       NumToString( lineNo, numString );
362:       WHILE length( numString ) < MaxLNDigits DO { Add leading zeros }
363:         insert( '0', numString, 1 );
364:       PrnString( numString ); PrnString( ': ' );
365:       lineNo := lineNo + 1
366:     END; { if }
367:   PrnString( s );
368:   PrnNewLine
369: END; { PrnLine }
370:
371:
372: FUNCTION PrnNew;
373:
374: { Return handle to a new, initialized, relocatable PrnSpec record. }
375:
376: BEGIN
377:   prnSpec := PrnHandle( NewHandle( SizeOf( PrnSpecRec ) ) );
378:   IF prnSpec <> NIL THEN WITH prnSpec^^ DO
379:     BEGIN { assign default values to record fields }
380:       lut := 2;
381:       laf := 1;
382:       cpl := 80;
383:       lpp := 66;
384:       ltp := 57;
385:       lsp := 0;
386:       tab := 8;
387:       spn := 1;
388:       sln := 1;
389:       header := '';
390:       footer := '';
391:       printHeader := FALSE;
392:       printFooter := FALSE;
393:       hasFormFeed := TRUE;
394:       lineNumbers := FALSE;
395:       textStyle := PrnNoStyle
396:     END;
397:   PrnNew := PrnSpec
398: END; { PrnNew }
399:
400:
401: PROCEDURE PrnDispose;
402:
403: { Dispose handle to printer spec record }
404:
405: BEGIN
406:   DisposHandle( Handle( pHand ) ); { Dispose of printer specs }
407:   pHand := NIL { Avoid dangling pointers }
408: END; { PrnDispose }
409:
410:
411: PROCEDURE PrnStart;
412:
413: { Initialize printer }
414:

```

```

415:  VAR
416:
417:      dateTime    : LONGINT;      { Encoded date and time }
418:      dateString  : Str255;      { Date and time as strings }
419:      timeString  : Str255;
420:
421:  BEGIN
422:      prnSpec := pHand;
423:      PrDrvOpen;                  { Open printer driver }
424:      PrCtlCall( iPrDevCtl, lPrReset, 0, 0 ); { Send reset command }
425:      WITH prnSpec^^ DO
426:          BEGIN
427:              pgLine := 0;          { Line number on page }
428:              column := 0;          { Column number on line }
429:              lineNo := sln;        { Starting line number }
430:              pageNo := spn;        { Starting page number }
431:              lutltp := lut + ltp;  { Avoids readding this }
432:              lutltplaf := lut + ltp + laf; { ditto above comment }
433:              PrnSetStyle( textStyle ); { Select printing style }
434:
435:              { NOTE: prnSpec^^ is invalid now due to possible heap compaction. }
436:
437:          END; { with }
438:          GetDateTime( dateTime );
439:          IUDateString( dateTime, shortDate, dateString );
440:          IUTimeString( dateTime, FALSE, timeString );
441:          dtpString := concat( dateString, ' ', timeString, ' Page-' );
442:          titleOverHead := length( dtpString ) + MaxPNDigits;
443:          lastLineDone := FALSE;
444:          PrnHeader
445:      END; { PrnStart }
446:
447:
448:  PROCEDURE PrnEnd;
449:
450:  { Deinitialize printer }
451:
452:  BEGIN
453:      PrnFooter;                  { Print footer on last page }
454:      PrDrvClose                  { Close printer driver }
455:  END; { PrnEnd }
456:
457:
458:  BEGIN
459:      prnSpec := NIL              { No print spec record in memory }
460:  END. { Unit }

```

ImageUnit Play-by-Play

Three data type declarations specify various printing features. **PrnStyles** (26–29) is an enumerated type whose elements each represent one printing style. **PrnNoStyle**, the first element, causes no change to the printer's current style. If you set switches inside your printer to select among its features, you can print with **PrnNoStyle** to use those settings instead of sending software codes to the printer to change styles. Table 7.1 lists the meaning of the other **PrnStyles** elements.

PrnSpecRec (34–57) is a large record that holds specifications to control printing style and page formatting. See the listing comments for the meaning of each field in this record. When you call function **PrnNew**, it creates a record of this type

Table 7.1 ImageWriter print styles from ImageUnit.

PrnStyles	Meaning	Characters Per Inch	Characters per 8-inch Line
PrnNoStyle	no effect	--	--
PrnExtended	extended	9	72
PrnPica	pica (standard)	10	80
PrnElite	elite	11.8	94
PrnPicaPro	pica proportional	10 (approx)	--
PrnElitePro	elite proportional	12 (approx)	--
PrnSemiCond	semi condensed	13.3	106
PrnCondensed	condensed	15	120
PrnUltraCond	ultra condensed	16.7	133
PrnHLStart	head line start	--	--
PrnHLEnd	head line end	--	--

on the heap and passes you a **PrnHandle** (31) to it. You can then use that handle to change the record's fields. For example, to set maximum characters per line to 132, you could write:

```
VAR ph : PrnHandle;

ph := PrnNew;
ph^.cpl := 132;
```

Local Declarations (77-281)

ImageUnit contains many local declarations and routines not directly available to your programs. The ASCII constants in lines 81-86 are the decimal values that represent control characters and a blank. The # symbol tells the compiler to consider the number as a character with that ASCII value. Change constants **MaxLNDigits** and **MaxPNDigits** (88-89) to the maximum number of digits in line numbers. A maximum of 2 allows numbers from 0 to 99, 3 permits up to 999, and so on.

Variable **prnSpec** (94) is a handle to the current printer specification record, with the formatting fields described earlier. String **dtpString** (96) holds the date, time, and page number string that the unit prints at the top of each new page if printer specification field **printHeader** is **TRUE**. Procedure **PrnStart** initializes this string. The seven integer variables and one Boolean at lines 98-105 control line and page numbers, and help position headers, footers, and text. These variables should have obvious meanings—see the Listing comments for details.

Procedure **ListChar** (107-119) sends a single character to the printer using a method called *text streaming*. This completely bypasses the printer driver logic that normally interprets bytes as commands to draw text and images dot-by-dot. Instead, **ListChar** sends bytes directly to the printer, which interprets them as ASCII character codes, helping the printer operate more quickly by reducing the amount of information it needs from the program.

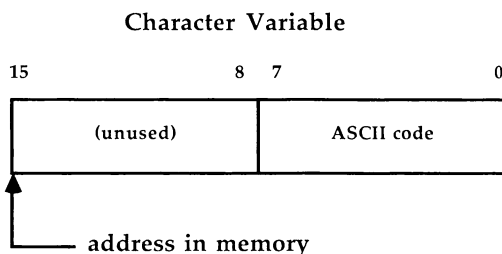


Figure 7.2 Because a character variable sits in the low order eight bits of a 16-bit word, the address of that word is one byte ahead of the character.

To accomplish this requires calling operating system routine **PrCtlCall** (Printer Control Call) (118), passing a constant (**iPrIOCtl**) that selects text streaming. (These toolbox identifiers are from the MacPrint unit, which ImageUnit uses at line 21.) The procedure moves character **ch** into the first byte of a two-byte buffer, passing the address of that buffer (and, therefore, the character) to **PrCtlCall**. It cannot pass the character address directly because characters are normally in the second eight bits of a 16-bit word—in other words, at the address of **ch** plus one byte as Figure 7.2 illustrates. The second parameter to **PrCtlCall**, 1, indicates the number of characters to print (118). The third parameter, 0, is required but meaningless.

PrnTab (122–137) correctly aligns columns with embedded tab control characters. This lets you print listings prepared with editors (not Turbo’s) that insert tabs to indent lines and align assembly language columns. Unless you change it, tabs are fixed at every eight columns.

PrnLf (140–148) and **PrnCr** (151–159) print line feeds and carriage returns. They also adjust variables **pgLine** and **column** to keep track of the exact print position on the page. A line feed advances the paper by a single line, keeping the print position at the same column. A carriage return moves the print position to the extreme left and does not advance the paper. Be sure to obey these rules when modifying ImageUnit to run other printers.

(Note: if printing is double-spaced, make sure your ImageWriter switches are set to *not* add line feeds automatically to carriage returns. Consult your printer manual for information about changing this setting.)

ImageUnit calls **PrnAdvanceLine** (162–174) for every new line, executing **n** line feeds, usually equal to the line spacing specification **lsp** (42). A zero value prints single-spaced text. One prints double-spaced lines (one extra blank line); two prints triple-spaced text, and so on.

PrnFf (177–189) advances paper to the top of a new page, or form. Set the lines per page field **lpp** (40) to adjust pages for different paper lengths. Also set field **hasFormFeed** (52) **TRUE** if your printer understands ASCII 12 as a command to advance to the top of the next form. If this value is **FALSE**, line 187 calls **PrnAd-**

vanceLine to advance the paper one line at a time, simulating form feeds on printers that don't understand them. (Most do.) If pages advance with jerky steps, or if headers do not print at the tops of new pages, you are probably using the wrong settings.

PrnFooter (192–221) and **PrnHeader** (224–253) have obvious jobs. They use your printer's headline style to print titles and footnotes, features you can turn on or off by changing specifications **printHeader** and **printFooter** (50–51). You are free to change these routines if you don't like ImageUnit's header style. If you do that, be sure to call **PrnFf** as the last step in **PrnFooter** (220) and advance to the first text line on a page as shown in **PrnHeader** (252). To print text, use **PrnString** or **ListChar**. To advance lines, call **PrnAdvanceLine**. Do not use other printing methods and don't issue your own carriage returns and line feeds, or page formatting may not work correctly.

The final two local routines, **PrnNewPage** (256–269) and **PrnNewLine** (272–281) cooperate to format pages into fixed numbers of lines with optional headers and footers. You probably should not change these two in any way. Notice that line 262 checks global **lastLineDone**. This avoids printing a blank page when the number of print lines exactly fit. Later, you'll learn how to set this variable.

Next are routines you can call from programs. Before diving into your own ImageUnit projects, though, read through the following descriptions. There are a few rules you should know about.

```
PROCEDURE PrnSetStyle( pStyle : PrnStyles );
```

Pass a **PrnStyles** element (27–29) to **PrnSetStyle** (284–315) to select one or another printer feature. In some cases, you can combine features to produce various effects. For example, pass **PrnCondensed** followed by **PrnHLStart** to print condensed headlines.

When modifying ImageUnit to run other printers, change this procedure to send the control characters your printer understands. For letter-quality printers, some of which have no special features, leave the procedure blank, removing lines 294–314. If you are blessed with more than one printer, you might also consider adding a method to select one or the other.

```
PROCEDURE PrnChar( ch : CHAR );
```

Call procedure **PrnChar** (318–332) to print a single character, which can have any ASCII value. It traps the control characters shown in the listing, calling local routines to handle tabs, carriage returns, line feeds, and form feeds. It passes other characters directly to **ListChar** (329).

Don't use **PrnChar** to select printing styles or various options by passing control codes. Instead, modify **PrnSetStyle** as the previous section describes and call it to change print styles.

```
PROCEDURE PrnString( s : Str255 );
PROCEDURE PrnLine( s : Str255; lastLine :
  Boolean );
```

PrnString (335–346) is similar to a Pascal **Write** statement. It prints a line of text, up to 255 characters at a time, starting at the current print position. It does not add a line feed or carriage return after printing. Therefore, you can call it several times in a row to print different items on the same line.

PrnLine (349–369) operates like a Pascal **WriteLn** statement. It calls **PrnString** (367) and then starts a new line (368). It also adds line numbers to the left column if **lineNumbers** (53) is **TRUE**. If you don't want leading zeros in these numbers, change character '0' in line 363 to a blank.

Parameter **lastLine** tells **PrnLine** if this is the last line you will print. If **TRUE**, then even if this is the final line on the page, **ImageUnit** will *not* start a new page after printing the line. Usually, set **lastLine** to the result of Pascal's **EOF** function for the file you are printing. The next program, **MacLister**, explains how to do this.

```
FUNCTION PrnNew : PrnHandle;
```

Call **PrnNew** (372–398) as the first step in your printing procedure or in your program's initialization section before calling any other **ImageUnit** routine. The function passes back a handle to a printer specification record, which it creates on the heap (377). Save this handle in a **PrnHandle** variable, which you'll then pass to **PrnStart** before printing. You can call **PrnNew** several times to create more than one printer specification in memory, saving the resulting handles in an array or in multiple variables.

Each new specification record has the default settings at lines 380–395. You may change any of these defaults now or later from inside your program. If you have a wide carriage (14-inch) printer, for example, you might want to change characters per line field **cpl** (382) to 132. That way, you won't have to reassign that value in every program that uses **ImageUnit**.

```
PROCEDURE PrnDispose( VAR pHand : PrnHandle );
```

After printing, pass your printer specification handle to **PrnDispose** (401–408), which removes the **PrnSpecRec** variable from the heap and sets handle **pHand** to **NIL**. Always dispose your handles when you are done with them. This frees the space they occupy, letting the memory manager reuse the memory for other purposes.

```
PROCEDURE PrnStart( pHand : PrnHandle );
```

After **PrnNew** initializes a new specifications record, call **PrnStart** (411–445) before printing the first character. This routine initializes various **ImageUnit**

variables, prepares the header and footer strings, and saves **pHand** as the current **PrnHandle**, which other routines will then use.

Lines 423–424 initialize the printer driver to prepare it to accept characters. **PrnDrvOpen** also loads the driver code into memory if it is not already there. **PrnCtlCall** sends a reset command to the printer, reconfiguring its settings to whatever they normally are when you first turn it on.

Lines 427–433 assign various starting values to variables that other procedures use. There's nothing that you can change here, but if you add any new assignments, be aware that calling **PrnSetStyle** (433) could cause the Memory Manager to rearrange relocatable objects on the heap, making the double dereference to **prnSpec^{^^}** (425) invalid afterwards.

The rest of **PrnStart** sets global **dtpString** to the title printed at the top of each new page. Line 439 converts the system date into a string. You can alter its format by changing the second parameter (now **ShortDate**) as follows:

```
ShortDate = 12/31/90
LongDate = Monday, December 31, 1990
AbbrevDate = Mon, Dec 31, 1990
```

Similarly, you can add seconds to the time string by changing parameter **FALSE** in line 440 to **TRUE**.

PROCEDURE PrnEnd;

After printing your last character, call **PrnEnd** (448–455) to finish the page, printing the final footer and advancing the paper. The procedure also closes the printer driver (454), a necessary step before calling **PrnStart** for another printout.

Using ImageUnit Tools

At this point, you may understand what the ImageUnit tools do but not have a clear picture of how to put them together. The following notes will help:

- Create a local **PrnHandle** variable, which I'll call **pHand** from now on although any name will do. Call **PrnNew** and assign its result to this variable. If its value is **NIL**, do not continue with the following steps.
- Initialize any printer specifications, assigning values to selected fields (see lines 37–55). For example, to change fixed tabs to 4, use the statement:

```
pHand^^.tab:=4;
```

Also set the **header** and **footer** strings (47–48) to your report title or file name. Skip these steps to use the default settings.

- Call **PrnStart**, passing **pHand** as a parameter. This tells ImageUnit to use your specifications, opens the printer driver, and initializes the printer.
- Print your document, calling any of the four procedures at lines 60–66 to print lines and characters and to change print styles. To advance the printer a line at a time, print a line feed character (ASCII 10) with **PrnChar**. To advance to the top of a new page, print a form feed character (ASCII 12). Page numbers, headers, and footers print automatically at the correct times and places. You don't have to call any other routines to make this happen.
- After printing the last character, call **PrnEnd** to close the printer driver, print the final footer, and advance the paper to the start of a new page.
- To start a new printout, call **PrnStart** using the same **pHand**. When you are completely done printing, call **PrnDispose** to dispose **pHand** and release the memory that the printer specification record occupies on the heap.

If your headers are too long and report titles bump into the date and time, there's an easy way to fix the problem. Add the following statement between lines 207–208 and also between 236–237:

```
PrnSetStyle( PrnUltraCond );
```

This changes the header and footer lines to the ImageWriter's most highly condensed style. If that solution doesn't please you, remove the time from headers to gain a little extra room. To do this, replace line 440 with the following statement. There are no blanks between the two single quotes.

```
timeString := '';
```

PUTTING YOUR TOOLS TO WORK

As a final example, MacLister uses most of the features described in this and in earlier chapters. The program is a source code lister that you can use to keep track of your printouts. It operates both short and wide carriage ImageWriters (or other printers if you modify ImageUnit), adds line numbers, and prints headers and footers. The header contains the date and time for reference, and the footer is in the lower right corner of the page, making it easy to find files by flipping through pages in printout binders.

Figure 7.3 shows MacLister's File menu commands. Open brings up the standard file dialog (not shown here). Choose a file to print and MacLister displays its first 24 lines in the program window. You cannot scroll these lines or modify them in any way—they are merely for reference, showing the file you are about to print. Start printing by choosing the Print command. You can print additional copies of the open file by choosing Print again.

File		
Open...	⌘O	- Open text file to print
Close		- Close desk accessory
Print	⌘P	- Print open file
Transfer	⌘T	- Transfer to other program
Quit	⌘Q	- Return to Finder

Figure 7.3 MacLister's File menu.

The Close command operates differently than usual. It closes an open desk accessory—not an open text file. To print different files, just choose Open again. This saves time by avoiding the unnecessary step of closing files to which you make no changes.

Choose the Transfer command to run a different application, for example Turbo Pascal. *Never choose Transfer when running MacLister from the Turbo Editor. The resulting crash will require you to reboot your computer.* In my system, I install MacLister in Turbo's own Transfer menu. This lets me transfer back and forth to print files without having to return to the Finder.

The Edit menu illustrated in Figure 7.4 contains the usual editing commands, which operate only for desk accessories. Choose Options to bring up the dialog in Figure 7.5. Click the options you want to use for subsequent printouts. Select wide carriage printing if your printer can handle 14-inch wide paper, or when using a compressed print style on 8-inch paper. You can change the dialog's default options by revising the program. Instructions follow in the play-by-play description.

Figure 7.6 shows MacLister's final menu, from which you select a printing style. As with the Options dialog, you can revise the program to change the default style, normally Pica. The figure lists the number of characters per inch (cpi) that each

Edit		
Undo	⌘Z	- Edit commands
Cut	⌘H	- " "
Copy	⌘C	- " "
Paste	⌘U	- " "
Clear		- " "
Options...		- Select MacLister options

Figure 7.4 MacLister's Edit menu.

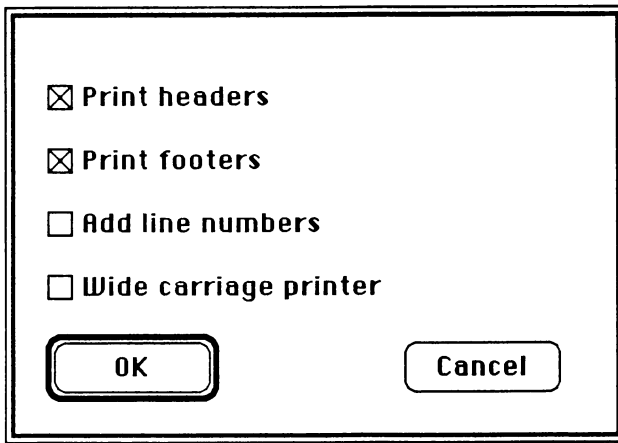


Figure 7.5 MacLister's Options dialog.

style gives. Some of these values are approximate and some do not match those in the ImageWriter manual. All are direct measurements of test printouts on an ImageWriter 14-inch printer. To select a different style, choose it from the menu. You'll see nothing happen on screen, but your selection will have a checkmark the next time you view the menu.

To compile the program, first type in Listing 7.8 and save as MACLISTER.R. Compile this text with RMaker to produce the program's resource file. Next, type in Listing 7.9 and save as MACLISTER.PAS. Before compiling with Turbo, you must have previously compiled units MacExtras, DialogUnit, ErrorUnit, ImageUnit, and Transfer. You should probably compile MacLister to a disk code file. You can run it from the Turbo editor but, if you do that, remember not to choose the Transfer command, which works correctly only if you run the program from the Finder or by transferring to it from another application.

Style	
Extended	- 9 cpi
✓ Pica	- 10 cpi
Elite	- 11.8 cpi
Pica proportional	- 10 cpi (approx.)
Elite proportional	- 12 cpi (approx.)
Semi-condensed	- 13.3 cpi
Condensed	- 15 cpi
Ultra-condensed	- 16.7 cpi

Figure 7.6 MacLister's Style menu.

Listing 7.8. MACLISTER.R

```

1: *-----*
2: * MacLister.PAS resources -- Compile with RMaker *
3: *-----*
4:
5: Programs:MacLister.F:MacLister.RSRC    ;; Send output to here
6:
7:
8: *-----*
9: * About box string list *
10: *-----*
11:
12: TYPE STR#
13:     ,1 (32)
14: 6
15: MacLister
16: by Tom Swan
17: Version 1.00
18: (C) 1987 by Swan Software
19: P. O. Box 206, Lititz, PA 17543
20: (717)-627-1911
21:
22:
23: *-----*
24: * The Apple Info menu *
25: *-----*
26:
27: TYPE MENU
28:     ,1
29: \14
30:     About MacLister...
31:     (-
32:
33:
34: *-----*
35: * The File menu *
36: *-----*
37:
38: TYPE MENU
39:     ,2
40: File
41:     Open... /O
42:     (Close
43:     (-
44:     (Print /P
45:     (-
46:     Transfer /T
47:     Quit /Q
48:
49:
50: *-----*
51: * The Edit menu *
52: *-----*
53:
54: TYPE MENU
55:     ,3
56: Edit
57:     (Undo /Z
58:     (-
59:     (Cut /X
60:     (Copy /C
61:     (Paste /V
62:     (Clear
63:     (-
64:     Options...
65:

```

```

66:
67: *-----*
68: * The Style menu *
69: *-----*
70:
71: TYPE MENU
72:     ,4
73: Style
74:     Extended
75:     Pica
76:     Elite
77:     Pica proportional
78:     Elite proportional
79:     Semi-condensed
80:     Condensed
81:     Ultra-condensed
82:
83:
84: *-----*
85: * The error alert template *
86: *-----*
87:
88: TYPE ALRT
89:     ,999 (4)           ;; Resource ID, (4) = preload
90:     45 28 154 481      ;; Top Left Bottom Right
91:     999                ;; Item list ID (following)
92:     5555               ;; Stages (none)
93:
94:
95: TYPE DITL           ;; Error alert item list
96:     ,999 (32)        ;; Resource ID , (32)=purgeable
97:     4
98:
99: BtnItem Enabled      ;; 1. Ok button
100: 71 360 103 440
101: Ok
102:
103: StatText Disabled    ;; 2. Error number
104: 25 80 41 165
105: Error ^0:
106:
107: StatText Disabled    ;; 3. Error message
108: 25 170 41 440
109: ^1
110:
111: StatText Disabled    ;; 4. Help message
112: 80 15 96 350
113: ^2
114:
115:
116: *-----*
117: * Window template *
118: *-----*
119:
120: TYPE WIND
121:     ,1000 (32)         ;; ID number and attribute (purgeable)
122:     untitled           ;; Window title
123:     46 7 328 502       ;; top, left, bottom, right coordinates
124:     Visible NoGoAway   ;; Visible window without close button
125:     4                  ;; Document window style without grow and zoom boxes
126:     0                  ;; Window reference (none)
127:
128:
129: *-----*
130: * Options dialog template *
131: *-----*
132:

```

(continued)

380 \equiv Programming with Macintosh Turbo Pascal

```
133: TYPE DLOG
134:     ,1001                ;; ID number
135: Options                  ;; Dialog title (not displayed)
136: 84 119 275 396          ;; Global coordinates for dialog window
137: Visible NoGoAway        ;; Make invisible with no close button
138: 1                        ;; Standard dialog box style
139: 0                        ;; Reference value (none)
140: 1001                     ;; Resource id of dialog item list
141:
142:
143: *-----*
144: * Options Dialog item list                                *
145: *-----*
146:
147: TYPE DITL
148:     ,1001 (32)           ;; Resource ID & attribute (purgeable)
149: 6                         ;; Number of items following
150:
151: BtnItem Enabled          ;; 1. OK button (must be first)
152: 151 15 175 89
153: OK
154:
155: BtnItem Enabled          ;; 2. Cancel button
156: 151 180 175 254
157: Cancel
158:
159: ChkItem Enabled          ;; 3. Program options
160: 25 10 45 270
161: Print headers
162:
163: ChkItem Enabled          ;; 4.
164: 55 10 75 270
165: Print footers
166:
167: ChkItem Enabled          ;; 5.
168: 85 10 105 270
169: Add line numbers
170:
171: ChkItem Enabled          ;; 6.
172: 115 10 135 270
173: Wide carriage printer
174:
175:
176: *-----*
177: * Printing in progress... dialog                          *
178: *-----*
179:
180: TYPE DLOG
181:     ,1002
182: Printing
183: 80 130 220 375
184: Visible NoGoAway
185: 1
186: 0
187: 1002
188:
189:
190: TYPE DITL                ;; Item list to dialog 1002
191:     ,1002 (32)
192: 3                          ;; Number of items in list
193:
194: StatText Enabled          ;; 1.
195: 33 20 56 220
196: Printing in Progress...
197:
198: StatText Disabled        ;; 2.
199: 71 20 88 235
200: To stop printing, hold down the
201:
```

```

202: StatText Disabled           ;; 3.
203: 87 20 104 235
204: \11 key and type period (.).      ;; \11 = command key symbol
205:
206:
207: * END

```

Listing 7.9. MACLISTER.PAS

```

1: {!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
2:
3: WARNING : Do not use the File menu Transfer command when compiling and
4: running MacLister directly from Turbo Pascal. The command works properly
5: only if you compile the program to disk and transfer to it or run it from
6: the Finder.
7:
8: {!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
9:
10:
11: {$O Programs:MacLister.F: }           { Send compiled code to here }
12: {$R Programs:MacLister.F:MacLister.Rsrc} { Use this compiled resource file }
13: {$U-}                                { Turn off standard library units }
14:
15:
16: PROGRAM MacLister;
17:
18: (*
19:
20: * PURPOSE : Pascal program (or any text file) listing printer
21: * SYSTEM   : Macintosh / Turbo Pascal
22: * AUTHOR   : Tom Swan
23:
24: *)
25:
26:
27: {$U Programs:Units.F:MacExtras }       { Open these library unit files }
28: {$U Programs:Units.F:DialogUnit }
29: {$U Programs:Units.F:ErrorUnit }
30: {$U Programs:Units.F:ImageUnit }
31: {$U Programs:Units.F:Transfer }
32:
33:
34: USES
35:
36:     PasInOut,
37:     Memtypes, QuickDraw, OSIntf, ToolIntf, PackIntf, MacExtras, MacPrint,
38:     DialogUnit, ErrorUnit, ImageUnit, Transfer;
39:
40:
41:
42: CONST
43:
44:     FileID      = 2;      { File menu Resource ID and commands }
45:     OpenCmd     = 1;
46:     CloseCmd    = 2;
47:     {-----}
48:     PrintCmd    = 4;
49:     {-----}
50:     TransferCmd = 6;
51:     QuitCmd     = 7;
52:
53:

```

(continued)

```

54:      (*EditID      = 3;*)      { Edit menu is defined in MacExtras }
55:      OptionsCmd    = 8;      { Additional command in edit menu }
56:
57:
58:      StyleID       = 4;      { Style menu }
59:      ExtendedCmd    = 1;      { Printer style selections }
60:      PicaCmd        = 2;
61:      EliteCmd       = 3;
62:      PicaProCmd     = 4;
63:      EliteProCmd    = 5;
64:      SemiCondCmd    = 6;
65:      CondensedCmd   = 7;
66:      UltraCondCmd   = 8;
67:
68:      DefaultStyle   = PicaCmd; { Or any Style Cmd number above }
69:
70:
71:      WindowID       = 1000; { Program window resource ID }
72:
73:      OptionsID      = 1001; { Options dialog resource ID }
74:      OptPrHeads     = 3;      { Option check boxes, corresponding to }
75:      OptPrFoots     = 4;      { their positions in the dialog item list }
76:      OptLineNum     = 5;
77:      OptWideCR      = 6;
78:
79:      PrintingID     = 1002; { Printing in progress dialog resource ID }
80:
81:
82:
83:  VAR
84:
85:      wRec           : WindowRecord;      { Program's window data record }
86:      wPtr           : WindowPtr;         { Pointer to above wRec }
87:
88:      fileOpen       : BOOLEAN;           { True if a file is open }
89:      fileName       : String[64];        { If fileOpen, this is its name }
90:      fileVar        : TEXT;              { If fileOpen, this is valid }
91:
92:      styleMenu      : MenuHandle;        { Printer styles menu }
93:      currentStyle   : INTEGER;           { Checked item in style menu }
94:
95:      printOptions   : ChecksRecord;      { Options set (see DialogUnit) }
96:
97:      quitRequested  : BOOLEAN;           { TRUE if quitting }
98:      transferring   : BOOLEAN;           { TRUE if transferring to next app }
99:
100:      nextApp        : SFReply;           { Next application if transferring }
101:
102:
103: PROCEDURE FixStyleMenu( CheckOnOff : BOOLEAN );
104:
105: { Make check mark in style menu agree with global currentStyle.
106: Add check mark if CheckOnOff is TRUE, otherwise remove it. }
107:
108: BEGIN
109:     CheckItem( StyleMenu, currentStyle, CheckOnOff )
110: END; { FixStyleMenu }
111:
112:
113: FUNCTION CmdToStyle( styleNum : INTEGER ) : PrnStyles;
114:
115: { Convert a Style menu number into an ImageUnit PrnStyle value }
116:

```

```

117: BEGIN
118:   CASE styleNum OF
119:     ExtendedCmd : CmdToStyle := PrnExtended;
120:     PicaCmd      : CmdToStyle := PrnPica;
121:     EliteCmd     : CmdToStyle := PrnElite;
122:     PicaProCmd   : CmdToStyle := PrnPicaPro;
123:     EliteProCmd  : CmdToStyle := PrnElitePro;
124:     SemiCondCmd  : CmdToStyle := PrnSemiCond;
125:     CondensedCmd : CmdToStyle := PrnCondensed;
126:     UltraCondCmd : CmdToStyle := PrnUltraCond;
127:     OTHERWISE    CmdToStyle := PrnNoStyle
128:   END { case }
129: END; { CmdToStyle }
130:
131:
132: PROCEDURE CloseFile;
133:
134: { Close the current file }
135:
136: BEGIN
137:   IF fileOpen THEN
138:     BEGIN
139:       Close( fileVar ); { Close Pascal file }
140:       fileOpen := FALSE { Reset global flag }
141:     END { if }
142:   END; { CloseFile }
143:
144:
145: PROCEDURE DoOpen;
146:
147: { Respond to File menu Open command. Okay to open another
  file when one is already open. }
148:
149: VAR
150:   reply : SFReply;
151:   errCode : INTEGER;
152:
153: BEGIN
154:   IF GetFileName( reply, 'TEXT' ) THEN
155:     WITH reply DO
156:       IF SetVol( NIL, vRefNum ) = noErr THEN
157:         BEGIN
158:           IF fileOpen
159:             THEN CloseFile;
160:           { $i- } Reset( fileVar, fName ); { $i+ }
161:           errCode := IoResult;
162:           IF errCode = 0 THEN
163:             BEGIN
164:               fileOpen := TRUE;
165:               fileName := fName; { Save file name }
166:               SetWTitle( wPtr, fileName ); { Change window title }
167:               EnableItem( fileMenu, CloseCmd ); { Fix up menus }
168:               EnableItem( fileMenu, PrintCmd );
169:               InvalRect( wPtr^.portRect ) { Force update event }
170:             END ELSE
171:               IoError( errCode, 'Check disk and try again' )
172:             END ELSE
173:               IoError( -53, 'Lost disk volume' ) {-53="Vol not on-line"}
174:           END; { DoOpen }
175:
176: PROCEDURE DoClose;
177:
178: { Respond to File menu Close command. Note: it is not possible
  to close the main program window. }
179:
180:
181:
182:
183:

```

(continued)


```

184: BEGIN
185:   IF FrontWindow <> wPtr
186:     THEN CloseDAWindow
187:   END; { DoClose }
188:
189:
190: PROCEDURE DoPrint;
191:
192: { Print text. Assume global fileVar is open. }
193:
194:   VAR
195:
196:     s      : Str255;
197:     prnSpec : prnHandle;
198:     dp      : DialogPtr;
199:
200:
201:   PROCEDURE InitPrnSpec;
202:
203:   { Initialize ImageUnit parameters. Other prnSpec fields keep
204:     default values defined in the unit. }
205:
206:     VAR
207:
208:       cpi : Real;    { Characters per inch }
209:
210:   BEGIN
211:     WITH prnSpec^^ DO
212:       BEGIN
213:
214:         header := fileName;
215:         footer := fileName;
216:         textStyle := CmdToStyle( CurrentStyle );
217:
218:         CASE CurrentStyle OF
219:           ExtendedCmd : cpi := 9;      { cpi = chars per inch }
220:           PicaCmd      : cpi := 10;    { exact }
221:           EliteCmd     : cpi := 11.8;  { Manual says 12 }
222:           PicaProCmd   : cpi := 10;    { approx. }
223:           EliteProCmd  : cpi := 12;    { approx. }
224:           SemiCondCmd  : cpi := 13.3;  { Manual says 13.4 }
225:           CondensedCmd : cpi := 15;    { exact }
226:           UltraCondCmd : cpi := 16.7  { Manual says 17 }
227:         END; { case }
228:
229:         WITH printOptions DO
230:           BEGIN
231:             printHeader := OptPrHeads IN selections;
232:             printFooter := OptPrFoots IN selections;
233:             lineNumbers := OptLineNum IN selections;
234:             IF OptWideCR IN selections      { Assign chars per line: }
235:               THEN cpl := trunc( cpi * 13.5 ) { for wide carriage }
236:             ELSE cpl := trunc( cpi * 8.0 ) { for standard carriage }
237:           END { with }
238:
239:         END { with }
240:       END; { InitPrnSpec }
241:
242:
243:   PROCEDURE FixUp( VAR s : Str255 );
244:
245:   { Remove embedded form feeds from string s. Optional, but some
246:     Macintosh example text files have these controls, which
247:     conflict with MacLister's paging. }
248:
249:   CONST
250:
251:     ff = #12;    { ASCII form feed control character }
252:

```

```

253:     VAR
254:
255:         p : INTEGER;
256:
257:     BEGIN
258:         p := pos( ff, s );
259:         WHILE p > 0 DO
260:             BEGIN
261:                 Delete( s, p, 1 );
262:                 p := pos( ff, s )
263:             END { while }
264:         END; { FixUp }
265:
266:
267: PROCEDURE DisplayDialog;
268:
269: { Display "Printing in progress..." dialog box }
270:
271:     VAR
272:
273:         b : BOOLEAN;
274:         i : INTEGER;
275:         dEvent : EventRecord;
276:
277:     BEGIN
278:         dp := GetNewDialog( PrintingID, NIL, POINTER(-1) );
279:         b := GetNextEvent( UpdateMask, dEvent );
280:         b := DialogSelect( dEvent, dp, i )
281:     END; { DisplayDialog }
282:
283:
284: PROCEDURE EraseDialog;
285:
286: { Remove the "Printing..." dialog box }
287:
288:     BEGIN
289:         IF dp <> NIL
290:             THEN DisposDialog( dp )
291:         END; { EraseDialog }
292:
293:
294: FUNCTION Finished : BOOLEAN;
295:
296: { TRUE if end of file reached or Command-period typed to end printing.
297:   Also call SystemTask to give DAS their fair share of time. }
298:
299:     VAR
300:
301:         kbdEvent : EventRecord;
302:         ch : CHAR;
303:
304:     BEGIN
305:         SystemTask;
306:         IF EOF( fileVAR )
307:             THEN
308:                 Finished := TRUE
309:             ELSE
310:                 IF GetNextEvent( KeyDownMask, kbdEvent )
311:                     THEN
312:                         WITH kbdEvent DO
313:                             BEGIN
314:                                 ch := CHR( BitAnd( message, charCodeMask ) );
315:                                 Finished := ( BitAnd( modifiers, CmdKey ) <> 0 ) AND
316:                                             ( ch = '.' )
317:                             END
318:                         ELSE
319:                             Finished := FALSE
320:                     END; { Finished }
321:

```

(continued)

```

322:
323:   BEGIN
324:     prnSpec := PrnNew;           { Allocate new PrnSpecs record }
325:     IF prnSpec <> NIL THEN
326:       BEGIN
327:         InitPrnSpec;             { Initialize printing parameters }
328:         PrnStart( prnSpec );     { Start printing first page }
329:         DisplayDialog;           { Display "printing..." message }
330:         Reset( fileVar );        { Reset to beginning of text file }
331:         WHILE NOT Finished DO
332:           BEGIN
333:             Readln( fileVar, s ); { Read one line from file }
334:             FixUp( s );           { Remove embedded form feeds }
335:             PrnLine( s, EOF( fileVar) ) { Print the line }
336:           END; { while }
337:         PrnEnd;                  { Tell PrintMaster we're done }
338:         PrnDispose( prnSpec );   { Dispose the prnSpec handle }
339:         EraseDialog              { Remove the "printing..." message }
340:       END { if }
341:     END; { DoPrint }
342:
343:
344: PROCEDURE DoTransfer;
345:
346: { Respond to File menu Transfer command. Actual transfer
347: occurs when program ends. }
348:
349:   BEGIN
350:     IF GetApplName( nextApp ) THEN
351:       BEGIN
352:         transferring := TRUE;
353:         quitRequested := TRUE
354:       END { if }
355:     END; { DoTransfer }
356:
357:
358: PROCEDURE DoOptions;
359:
360: { Display options dialog and change print options }
361:
362:   VAR
363:
364:     dp : DialogPtr;
365:     itemHit : INTEGER;
366:     checks : ChecksRecord;
367:
368:   BEGIN
369:     checks := printOptions;      { Copy current options }
370:     dp := GetNewDialog( OptionsID, NIL, Pointer(-1) );
371:     IF dp <> NIL THEN
372:       BEGIN
373:         OutLineOK( dp );
374:         InitChecks( dp, checks );
375:         REPEAT
376:           ModalDialog( NIL, itemHit );
377:           IF ( itemHit <> Ok ) AND ( itemHit <> Cancel )
378:             THEN CheckBox( dp, checks, itemHit )
379:           UNTIL ( itemHit = Ok ) OR ( itemHit = Cancel );
380:           IF itemHit = Ok
381:             THEN printOptions := checks;
382:           DisposDialog( dp )
383:         END { if }
384:       END; { DoOptions }
385:
386:

```

```

387: PROCEDURE DoFileMenuCommands( cmdNumber : INTEGER );
388:
389: { Execute command in the File menu }
390:
391: BEGIN
392:     CASE cmdNumber OF
393:         OpenCmd      : DoOpen;
394:         CloseCmd     : DoClose;
395:         PrintCmd      : DoPrint;
396:         TransferCmd   : DoTransfer;
397:         QuitCmd       : quitRequested := TRUE
398:     END { case }
399: END; { DoFileMenuCommands }
400:
401:
402: PROCEDURE DoEditMenuCommands( cmdNumber : INTEGER );
403:
404: { Execute command in the Edit menu }
405:
406: BEGIN
407:     IF NOT SystemEdit( cmdNumber - 1 ) THEN
408:         IF cmdNumber = OptionsCmd
409:             THEN DoOptions
410:         END; { DoEditMenuCommands }
411:
412:
413: PROCEDURE DoStyleMenuCommands( cmdNumber : INTEGER );
414:
415: { Select a printing style }
416:
417: BEGIN
418:     FixStyleMenu( FALSE );           { Remove old check mark }
419:     currentStyle := cmdNumber;       { Change print style setting }
420:     FixStyleMenu( TRUE )             { Add new check mark }
421: END; { DoStyleMenuCommands }
422:
423:
424: PROCEDURE DoCommand( command : LongInt );
425:
426: { Execute a menu command }
427:
428: VAR
429:
430:     whichMenu   : INTEGER;           { Menu number of selected command }
431:     whichItem   : INTEGER;           { Menu item number of command }
432:
433: BEGIN
434:
435:     whichMenu := HiWord( command );  { Find the menu }
436:     whichItem := LoWord( command );  { Find the item }
437:
438:     CASE whichMenu OF
439:
440:         AppleID   : DoAppleMenuCommands( whichItem );
441:         FileID    : DoFileMenuCommands( whichItem );
442:         EditID    : DoEditMenuCommands( whichItem );
443:         StyleID   : DoStyleMenuCommands( whichItem )
444:
445:     END; { case }
446:
447:     HiliteMenu( 0 ) { Unhighlight menu title }
448:
449: END; { DoCommand }
450:
451:

```

(continued)

```

452: PROCEDURE ListFile;
453:
454: { Display a few lines from open file. Assume file is open and
455:   wPtr is the current window. }
456:
457:   VAR
458:
459:       s : Str255;
460:       y : INTEGER;
461:
462:   BEGIN
463:       Reset( fileVar );      { Reset to top of file }
464:       y := 0;
465:       WHILE ( y < 24 ) AND ( NOT EOF( fileVar ) ) DO
466:           BEGIN
467:               ReadLn( fileVar, s );
468:               MoveTo( 10, 15 + ( y * 11 ) );
469:               DrawString( s );
470:               y := y + 1
471:           END { for }
472:       END; { ListFile }
473:
474:
475: PROCEDURE DrawContents( whichWindow : WindowPtr );
476:
477: { Display window contents }
478:
479:   BEGIN
480:       EraseRect( whichWindow^.portRect );
481:       IF fileOpen AND ( whichWindow = wPtr )
482:           THEN ListFile
483:       END; { DrawContents }
484:
485:
486: PROCEDURE MouseDownEvents;
487:
488: { Someone pressed the mouse button. Check its location and respond. }
489:
490:   VAR
491:
492:       partCode : INTEGER;      { Identifies what item was clicked. }
493:
494:   BEGIN
495:
496:       WITH theEvent DO
497:
498:           BEGIN
499:
500:               partCode := FindWindow( where, whichWindow );
501:
502:               CASE partCode OF
503:
504:                   inMenuBar
505:                       : DoCommand( MenuSelect( where ) );
506:
507:                   inSysWindow
508:                       : SystemClick( theEvent, whichWindow );
509:
510:                   inContent, inDrag
511:                       : IF whichWindow <> FrontWindow
512:                           THEN SelectWindow( whichWindow );
513:
514:                   END { case }
515:
516:               END { with }
517:
518:           END; { MouseDownEvents }

```

```

519:
520:
521: PROCEDURE KeyDownEvents;
522:
523: { A key was pressed. Do something with incoming character. }
524:
525:   VAR
526:
527:     ch : CHAR;
528:
529:   BEGIN
530:     WITH theEvent DO
531:       BEGIN
532:         ch := CHR( BitAnd( message, charCodeMask ) ); { Get character }
533:         IF BitAnd( modifiers, CmdKey ) <> 0 { If command key pressed }
534:           THEN DoCommand( MenuKey( ch ) ) { then execute command }
535:         END { with }
536:       END; { KeyDownEvents }
537:
538:
539: PROCEDURE UpdateEvents;
540:
541: { Part or all of a window requires redrawing }
542:
543:   VAR
544:
545:     oldPort : GrafPtr; { For saving / restoring port }
546:
547:   BEGIN
548:     GetPort( oldPort ); { Save current port }
549:     whichWindow :=
550:       WindowPtr( theEvent.message ); { Extract window pointer }
551:     SetPort( whichWindow ); { Change current grafPort }
552:     BeginUpdate( whichWindow ); { Calculate new visRgn }
553:     DrawContents( whichWindow ); { Draw/redraw window contents }
554:     EndUpdate( whichWindow ); { Reset original visRgn }
555:     SetPort( oldPort ); { Restore old port }
556:   END; { UpdateEvents }
557:
558:
559: PROCEDURE ActivateEvents;
560:
561: { Activate or deactivate windows }
562:
563:   BEGIN
564:     WITH theEvent DO
565:       BEGIN
566:
567:         whichWindow := WindowPtr( message ); { Extract window pointer }
568:         SetPort( whichWindow ); { Change current port }
569:
570:         IF BitAnd( modifiers, activeFlag ) <> 0
571:           THEN FixEditMenu( FALSE ) { Activate a window }
572:           ELSE FixEditMenu( TRUE ) { Deactivate a window }
573:         END { with }
574:       END; { ActivateEvents }
575:
576:
577:
578: PROCEDURE SetUpMenuBar;
579:
580: { Initialize and display menu bar }
581:

```

(continued)

```

582:   BEGIN
583:
584:       appleMenu := GetMenu( AppleID ); { Read menu resources }
585:       fileMenu  := GetMenu( FileID );
586:       editMenu  := GetMenu( EditID );
587:       styleMenu := GetMenu( StyleID );
588:
589:       InsertMenu( appleMenu, 0 );      { Insert into menu list }
590:       InsertMenu( fileMenu,  0 );
591:       InsertMenu( editMenu,  0 );
592:       InsertMenu( styleMenu, 0 );
593:
594:       AddResMenu( appleMenu, 'DRVr' ); { Add desk accessory names }
595:
596:       DrawMenuBar                    { Display the menu bar }
597:
598:   END; { SetUpMenuBar }
599:
600:
601: PROCEDURE SetUpWindow;
602:
603: { Open program window, which remains open throughout program }
604:
605:   BEGIN
606:       wPtr := GetNewWindow( WindowID, @Wrec, POINTER( -1 ) );
607:       IF wPtr = NIL
608:       THEN ExitToShell;
609:       SetPort( wPtr );
610:       TextFont( Monaco );
611:       TextSize( 9 )
612:   END; { SetUpWindow }
613:
614:
615: PROCEDURE SetUpOptions;
616:
617: { Initialize printing options and other default settings.
618:   Assumes menu is initialized. }
619:
620:   BEGIN
621:       WITH printOptions DO          { Assign printing options }
622:       BEGIN
623:           firstCheck := OptPrHeads;
624:           lastCheck  := OptWideCR;
625:           selections := [ OptPrHeads, OptPrFoots ] { defaults }
626:       END; { with }
627:       currentStyle := DefaultStyle; { Assign printing style }
628:       FixStyleMenu( TRUE )          { Add check mark to style menu }
629:   END; { SetUpOptions }
630:
631:
632: PROCEDURE Initialize;
633:
634: { Program calls this routine one time at start }
635:
636:   BEGIN
637:       SetUpMenuBar;          { Initialize and display menus }
638:       SetUpWindow;           { Initialize program window }
639:       SetUpOptions;          { Initialize print options }
640:       fileOpen := FALSE;     { No file open yet }
641:       quitRequested := FALSE; { TRUE on selecting Quit command }
642:       DisplayAboutBox        { Identify program }
643:   END; { Initialize }
644:
645:
646: FUNCTION QuitConfirmed : BOOLEAN;
647:
648: { The program's "deinitialization" routine. In this program, it's
649:   okay to quit at anytime. }
650:

```

```

651:   BEGIN
652:     IF quitRequested
653:       THEN CloseFile;    { Close file if open }
654:       QuitConfirmed := quitRequested
655:     END; { QuitConfirmed }
656:
657:
658: PROCEDURE DoSystemTasks;
659:
660: { Do operations at each pass through main program loop }
661:
662:   BEGIN
663:
664:     SystemTask;    { Give DAs their fair share of time }
665:
666:     IF FrontWindow <> wPtr THEN
667:
668:       BEGIN { Set up menu commands for active desk accessory }
669:
670:         FixEditMenu( TRUE );
671:         EnableItem( fileMenu, CloseCmd );
672:         DisableItem( fileMenu, OpenCmd );
673:         DisableItem( fileMenu, PrintCmd );
674:         DisableItem( editMenu, OptionsCmd );
675:
676:       END ELSE
677:
678:       BEGIN { Set up menu commands for program window }
679:
680:         FixEditMenu( FALSE );
681:         EnableItem( fileMenu, OpenCmd );
682:         DisableItem( fileMenu, CloseCmd );
683:         IF fileOpen
684:           THEN EnableItem( fileMenu, PrintCmd );
685:         EnableItem( editMenu, OptionsCmd );
686:
687:       END { else }
688:
689:     END; { DoSystemTasks }
690:
691:
692: BEGIN
693:
694:   Initialize;
695:
696:   REPEAT
697:
698:     DoSystemTasks;
699:
700:     IF GetNextEvent( everyEvent, theEvent ) THEN
701:
702:       CASE theEvent.what OF
703:
704:         MouseDown    : MouseDownEvents;
705:         KeyDown      : KeyDownEvents;
706:         UpdateEvt    : UpdateEvents;
707:         ActivateEvt  : ActivateEvents
708:
709:       END { case }
710:
711:     UNTIL QuitConfirmed;
712:
713:     IF transferring
714:       THEN RunProgram( nextApp )
715:
716:   END.

```


MacLister Play-by-Play

MACLISTER.R (1-207)

There are few new elements in the resource text file and you should have little trouble understanding its definitions. Line 204, though, may appear strange. The \11 stands for the ASCII character with that hexadecimal value, the Command key symbol, which the program displays in its “Printing in Progress” dialog box (Figure 7.7). You can enter other ASCII codes in static text items this way. For example, \0D stands for a carriage return, \14 the Apple symbol, and so on.

MACLISTER.PAS (1-100)

Assign to **DefaultStyle** (68) any one of the constants in lines 59-66 to change the default printing style. Whichever constant you use, MacLister checks off that choice in the Style menu. To print 132-column listings on 8-inch paper, set **DefaultStyle** to **UltraCondCmd**.

Variable **nextApp** (100) holds the file name and volume reference number of the next application to run if you choose the program’s Transfer command. Instead of transferring directly to another program at that time, MacLister sets variable **transferring** (98) **TRUE**. Later (see lines 713-714), it calls **RunProgram** to run another application, letting the program first close any open files and perform other cleanup chores.

FixStyleMenu to DoClose (103-187)

FixStyleMenu (103-110) shows how to add a check mark to a menu choice. It calls **CheckItem** (109), passing the menu handle (**StyleMenu**), the item number (**currentStyle**) representing the position of the command in the pull-down menu

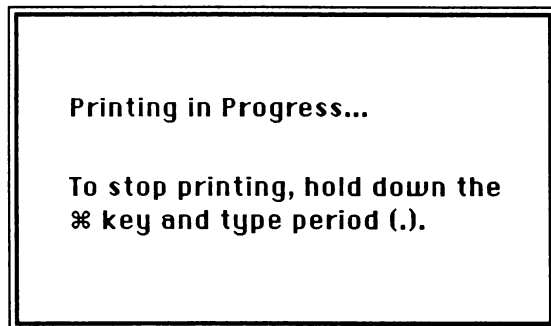


Figure 7.7 While printing, MacLister displays the message in this dialog.

window, and a Boolean value (**CheckOnOff**). If this value is **TRUE**, the routine adds a check mark; otherwise, it removes it. When you choose a different printing style, MacLister calls **FixStyleMenu** twice—once to remove the current check mark and once to add a new one.

Function **CmdToStyle** (113–129) takes a style number (one of the constants at lines 59–66) and returns the corresponding ImageUnit **PrnStyles** element, which the program then assigns to the printer specification record before each new print-out. This is the way MacLister recognizes your Style menu choice.

CloseFile (132–142) and **DoOpen** (145–176) both respect and properly set variable **fileOpen** to indicate whether a file is now open and on display. They use standard Pascal file techniques to open and close global variable **fileVar**. Notice that **DoOpen** closes **fileVar** (160–161) if it is now open. This lets you open new files without having to close another beforehand.

Line 168 sets the window title to the name of the open file. After that, the procedure invalidates the entire window (171) to force an update event, which eventually will display sample lines from the file.

DoClose (179–187) differs from previous procedures of the same name by closing only desk accessories. Although the program never calls this procedure unless a desk accessory's window is frontmost, line 185 checks this fact just to be safe before calling MacExtras tool **CloseDAWindow**.

DoPrint (190–341)

DoPrint is the workhorse procedure that prints the currently open file. It contains several sub procedures and a function that divide the routine into pieces. Procedure **InitPrnSpec** (201–240) assumes that variable **prnSpec** (197) is initialized to a printer specification handle. Lines 214–215 set both the header and footer titles to the current file name, printing this text at upper left and lower right on each page. Line 216 assigns the current printing style to field **textStyle**.

The **CASE** statement (218–227) sets a temporary variable **cpi** to the characters per inch for each printing style. It does this in order to accurately calculate printer specification field **cpl** (characters per line) at the later **IF** statement (234–236). This correctly sets that field depending on whether you select Wide Carriage with the Edit menu's Options command. Similarly, lines 231–233 set Boolean fields according to other options you select.

Procedure **FixUp** (243–264) is optional and you can remove it if you prefer. I include it to print text files that float around in Macintosh programming circles complete with annoying embedded form feed control characters to advance to new pages, frequently at the start of every new procedure. **FixUp** removes these controls, letting MacLister and ImageUnit format pages as they normally do. If you want the program to recognize embedded form feeds, remove procedure **FixUp** along with line 334, which calls it.

DisplayDialog (267–281) illustrates a method to display a kind of half-modeless dialog window. This procedure shows the "Printing in Progress" message in Figure

7.7. Line 278 opens and displays the dialog box in the usual way. Unfortunately, doing this does not display static text items—the messages inside the window—but only clickable buttons, check boxes, and radio buttons. The reason this occurs is because the dialog manager displays static text by invalidating its enclosing rectangle, generating an update event, which is not handled until you later call **ModalDialog**, as previous examples do. To display the text, lines 279–280 extract this update event by calling **GetNextEvent** with **UpdateMask** and then passing the resulting **dEvent** record to **DialogSelect**, which responds by drawing the text inside the window. Boolean variable **b** is a throw-away—its value is unimportant. To see why these steps are necessary, remove lines 279–280 and print a test file. You’ll see the dialog window but no text inside. The next procedure, **EraseDialog** (284–291), removes the dialog window disposing pointer **dp** before **DoPrint** ends.

Function **Finished** (294–320) has three jobs: to call **SystemTask** (305), giving desk accessories their fair share of time during printing; to test whether the program has reached the end of the text file; and to check for Command-period keypresses that interrupt printing.

After initializing **prnSpec** (324), the main **DoPrint** statements (327–339) read and print each line of text from the open file. Line 330 resets the file to start printing from the first line. Notice how line 335 both prints a line and tells **ImageWriter** whether this is the last line by passing the result of Boolean function **EOF**. (See the comments to procedure **PrnLine** in the **ImageUnit** play-by-play description.)

DoTransfer to DoCommand (344–349)

DoTransfer (344–355) sets global variables **transferring** and **quitRequested** to **TRUE** if **GetApplName** returns **TRUE** (350). If so, record variable **nextApp** contains the file name and volume number of the program that **MacLister** will run when it ends. The actual transfer occurs later (see lines 713–714).

By now you should be familiar with all of the programming in **DoOptions** (358–384). Likewise, you should have no trouble understanding the four procedures at lines 387–449, all of which respond to pull-down menu commands.

ListFile to END (452–716)

ListFile (452–472) responds to update events by way of the next procedure, **DrawContents**. It displays up to 24 reference lines from the open text file, using **MoveTo** and **DrawString** for each line. This method is inferior to the text tools in Chapter 5, but it works well enough for **MacLister**’s purposes. For a different text style or point size, change lines 610 and 611. If you modify the point size, you’ll have to adjust the expression in **MoveTo** (468) to separate one line from another.

To change the program’s default options (see Figure 7.5), modify line 625. Put the options you want inside the square brackets, using any combination of the four option constants (74–77).

Bibliography

SOFTWARE

HeapShow. B/T Computing Corporation, P.O. Box 1465, Euless, TX 76039. HeapShow is a desk accessory that graphically displays the contents of the heap while a program runs. It lets you peer into memory, examining your program's memory usage. Instructive, especially if you're just learning about memory management techniques.

TMON. Icom Simulations, Inc., 626 S. Wheeling Road, Wheeling, IL 60090. TMON (The Monitor) is a debugger that lists and changes memory, disassembles 68000 machine language, and lets you examine processor registers. If you are a casual programmer, you probably can use a less capable debugger such as MacsBug, which comes with Turbo Pascal. But professionals will appreciate TMON's many features.

Turbo Pascal Macintosh. Borland International Inc., 4585 Scotts Valley Drive, Scotts Valley, CA 95066. You need to have this product to type in and run all the examples in this book.

BOOKS

Apple Computer Inc. *Inside Macintosh, Vols 1, 2, 3, & 4*. Addison-Wesley, 1985, 1986. This four volume set is indispensable for Macintosh programmers. At least pick up volume 1—it describes many details referred to in this book. For serious programming, though, you'll need the entire set.

Chernicoff, Stephen. *Macintosh Revealed, Vols 1 & 2*. Hayden, 1985. Offers complete coverage of the Macintosh toolbox. Many descriptions and diagrams, but few working programming examples, make this reference less useful than it could be. The editor in volume 2 requires some revision before it will run in Turbo Pascal. For serious programmers only.

Knaster, Scott. *How to Write Macintosh Software*. Hayden, 1986. A not-for-beginners workbook of Macintosh programming. Contains many examples that should require minimal changes to run in Turbo Pascal. The chapter on debugging makes it worthwhile reading.

Swan, Tom. *Mastering Turbo Pascal*. Hayden Books-Howard W. Sams, 1986. The author's tutorial on programming Turbo Pascal for the IBM PC and CP/M computers. Contains many textbook examples that run in Turbo's dumb terminal window, described in Chapters 1 and 2.

...the ... of ...

THE ...

...the ... of ...

...the ... of ...

...the ... of ...

...the ... of ...

...the ... of ...

...the ... of ...

Index

- Absolute value, 63
- Activate event, 151
- Alert, 245, 259, 288, 290, 323. *See also*
 - Resource, alert
 - icons, 260
 - item list, 264–265, 294
 - preloading, 264
- ANIMATE.PAS, 101–103
- Animation, 93, 95, 104, 106–109, 116
- Application, 122, 128
- APSHELL.PAS, 128–136, 163
- APSHELL.R, 158–159

- BackPat, 89
- Backup file, 34
- Bic mode, 93
- Bit-map, 95–96, 100–101, 108
 - compared to bitMap record, 107
 - size in memory, 97
- Bit-mapped display, 56
- BtnItem, 271, 283
- Button, 60, 116, 257, 316
- BUTTONS.PAS, 254–256
- BUTTONS.R, 252–253

- CalcControlRects, 165, 167, 176
- Carriage return, 87
- CenterString, 165, 168, 177
- Character set, 28
- Character size, 84
- CHARS.PAS, 84–86, 89
- CharWidth, 84
- Check box, 279, 284, 328–329, 331, 394
- CheckBox, 325, 328, 331
- CheckOn, 324–325, 329
- ChkItem, 283
- Circle, 90
- ClearEol, 52
- ClearScreen, 52, 62
- Clipping, 117, 210
 - Clipping region, 73, 216
 - ClipRect, 75
 - Close, 31
 - Close box, 147, 155, 173, 178, 201
 - CloseDAWindow, 165, 167, 176, 393
 - CloseDialog, 258
 - ClosePicture, 216
 - CloseWindow, 143
 - Command key, 147, 149, 161–162, 321, 392
 - Compiler directive, 6, 9, 28, 31, 43, 136
 - Concat, 38, 106, 288
 - Control character, 52–53, 88, 370, 393
 - Control key, 30
 - Coordinate, 65, 79, 100
 - Coordinate plane, 64, 69, 77, 84, 97, 108, 118, 210
 - centering, 117
 - points on grid, 66
 - subtracting points, 67
 - Coordinate system, 73, 95–96, 101
 - Copy mode, 93
 - CopyBits, 95, 350
 - COS, 119
 - CountAppFiles, 48
 - Cursor, 76
 - changing shape, 126, 236, 318, 320, 322–323
 - handle, 322
 - hot spot, 146
 - style, 317
 - Cut and paste, 296, 321

- Data entry, 272, 295, 316, 319–320, 323
- DATAENTRY.PAS, 301–315
- DATAENTRY.R, 296–300
- Date, 210, 374
- Deactivate event, 151
- Debugging, 4, 6, 11
 - with comments, 59
 - with Pause, 171
 - with printer, 137

- DeleteLine, 44
- Destination rectangle, 241
- DETAB.PAS, 35–37
- Dialog, 245, 252, 277, 321. *See also* Resource,
 - dialog; Standard file dialog
 - adding features to window, 272
 - centering window, 329
 - creating, 258
 - data entry, 273
 - displaying, 257
 - edit areas, 317
 - item list, 252, 257, 259, 271, 277–278, 283, 316, 323, 329
 - modal, 245, 329
 - modeless, 245, 295, 317, 321, 323, 393
 - pointer, 272, 331
 - record, 259
 - resource, 323
 - static text, 394
 - template, 294
 - title, 256
 - window, 278, 294, 318–320, 322, 330–331, 339
- Dialog event, 321
- DialogSelect, 321
- DIALOGUNIT.PAS, 246, 324–328
- DisableMenu, 164–165, 173
- Display
 - changing modes, 89, 92
 - clearing, 89
 - conventional terminal, 55
 - initializing for graphics, 73
 - memory address, 100
 - width and height, 101
- Display Handler, 123–125, 127, 144, 152, 171, 361
- DisplayAboutBox, 165, 168, 177
- DisplayError, 286, 288
- Dispose, 47
- DisposeDialog, 259
- DisposeRgn, 100
- DisposeWindow, 143
- DisposHandle, 194
- DoAppleMenuCommands, 165, 169, 178
- DragTheWindow, 165, 173
- DrawIcon, 345, 348, 353
- Drawing modes, 93
- Dynamic variable, 185

- EditText, 316, 319, 323
- EnableMenu, 164–165, 172
- Endpoint paranoia, 67
- ENTRY.PAS, 274–277
- ENTRY.R, 273–274
- EOF, 30, 372, 394
- EraseOval, 90
- EraseRect, 89
- Error message, 245, 264, 285, 288–289, 320

- ERRORUNIT.PAS, 286–287
- ERRTEST.PAS, 292–294
- ERRTEST.R, 290–291
- Event, 127
 - flushing, 179
 - queue, 76, 129
 - record, 77, 242
- Event Handler, 124–125, 127–128, 142, 145–147, 150–151, 154, 170, 339
- Event-driven program, 121–122, 127–128, 329
- EventRecord, 170
- Exit, 63
- ExitToShell, 323

- File. *See also* Text file
 - checking for existence, 28
 - closing, 29
 - temporary, 34, 38
- File name, 236, 246, 250–252, 288, 330, 392–394
- File type, 251
- File variable, 49
- FillChar, 318
- FillOval, 90
- FillRect, 90
- FillRgn, 100
- FindControl, 243
- FindWindow, 146
- FixEditMenu, 165, 173
- FlushEvents, 76
- Font
 - base line, 86–87
 - height, 86, 177
 - menu, 17
 - names, 87
- Forwd, 59, 62
- FRACTAL.PAS, 111–115
- FrameOval, 90, 100
- FrameRect, 75
- Free union, 67, 69
- FrontWindow, 143
- FSOpen, 244
- FSRead and FSWrite, 43

- GetApplFiles, 48
- GetApplName, 337–339, 343
- GetCursor, 236
- GetDItem, 277, 294, 318–319, 323, 329
- GetEOF, 46
- GetFileName, 325, 327, 330
- GetFPos, 46
- GetIText, 278, 319
- GetMouse, 351
- GetNewDialog, 257–258, 323, 331
- GetNextEvent, 77, 116, 127, 154, 321, 394
- GetPen, 51
- GetPenState, 77–78
- GetPixel, 119

- GetPort, 76
- GetPortSize, 165, 167, 176
- GetResource, 350
- GetTUHandle, 238
- GetVol, 45
- Global Declaration, 122, 140
- Global variable, 72–73
- GrafPort, 70, 72, 76, 89, 101, 139, 145, 176
- GrafPtr, 75–76, 139, 145
- GRAPHSHELL.PAS, 71
- Grow box, 151–152, 173, 176, 235
 - shudder problem, 152
- Halt, 27
- Handle, 183, 186, 189, 191–192, 202, 211, 236–238, 278
 - creating, 190
 - dereferencing, 192–194, 239–240
 - dialog edit items, 318–319
 - disposing, 100, 194
- HCenter, 325–326, 329
- Heap, 11, 181, 192, 237, 323
 - diagram, 182
 - fragmenting, 72, 194, 202–203, 238, 257–258, 265, 323
 - grafPort, 72–73
 - locking, 193–194, 243–244
 - memory available, 19
 - space for objects, 191
- HideCursor, 63
- HiWord, 53
- Home, 59
- Horizontal axis, 65
- I/O error, 31, 35, 285, 320
- IBM PC, 21, 41–42, 46, 49, 52
 - command line parameters, 48
 - converting inLine statements, 46
 - files, 50
 - pointer variables, 51
 - pseudo windows, 51
- Icon, 93, 344
 - animating, 360
 - designing, 349
 - drawing, 361
 - image and mask, 349
 - initializing, 361
 - moving, 353
 - resource, 350, 360
- IconDragged, 345, 347, 351
- ICONTEST.PAS, 356–360
- ICONTEST.R, 353–355
- ICONUNIT.PAS, 344–349
- ImageUnit tools, 374–375
- IMAGEUNIT.PAS, 362–369
- ImageWriter, 362, 371, 375, 377, 394. *See also* Printer
- In-line machine language, 339–340
- Include directive, 12–13, 18
- InitButtons, 325, 327, 331
- InitChecks, 325, 328, 331
- InitCursor, 76, 318
- InitGraf, 75
- Initialization, 126
- InitNewIcon, 345–346, 350
- InRange, 164–165, 171, 329
- InsertLine, 46
- InsetRect, 75, 88–89
- InvalidRect, 244
- InvertOval, 90
- InvertRect, 89
- IoError, 286–287, 289, 320
- IoResult, 28–29, 31, 35, 285, 288–290, 320
- IsDialogEvent, 321
- Key code, 147–148, 317
- Keyboard event, 172
- Keypress, 63, 99
- LAUNCHER.PAS, 341–343
- LAUNCHER.R, 340–341
- LineTo, 79
- LONGINT, 46, 53
 - ordinal value, 51
- LoWord, 53
- MACEXTRAS.PAS, 163–170
- Machine language. *See* In-line machine language
- MACLISTER.PAS, 381–391
- MACLISTER.R, 378–381
- MacPaint file, 77, 116
- MACSTAT.PAS, 203, 205–209
- MACSTAT.R, 203–205
- MakeFileName, 325–326, 330
- Master pointer, 186–189, 191–192
- Memory block, 184–185, 239
 - non-relocatable, 185, 190
 - purgeable, 186
 - relocatable, 186–187, 216, 241, 244, 323
- MemTypes, 42, 49
- Menu, 172–173
 - handle, 170, 172, 317, 392
 - resource. *See* Resource, menu
- Menu Manager, 142, 144, 161
- ModalDialog, 258, 272, 278, 284, 294, 330–331, 394
- Modifier key value, 149
- MoreMasters, 189–190
- Mouse button, 63, 119, 171
- Mouse click, 109, 116
 - in icon, 351
 - in scroll bar, 235
 - in window, 146
- Mouse event, 242
- Mouse pointer, 76, 145, 243, 257, 271, 317, 322
 - coordinate, 351
- MoveIcon, 345, 348, 353

- MoveLeft and MoveRight, 47, 53
- MoveTo, 79
- MULTIWIND.PAS, 196–200
- MULTIWIND.R, 195
- New, 47
- NewHandle, 191
- NewWindow, 201, 203
- NUM.PAS, 8
- NUMBER.PAS, 22, 24–26
- NumToString, 50–51, 318
- OffsetRect, 88–89, 108
- OpenDeskAcc, 179
- OpenPicture, 216
- OpenPort, 73, 75–76
- Option key, 150
- OPTIONS.PAS, 280–283, 328, 331
- OPTIONS.R, 279–280
- Or mode, 93
- Ord4, 51
- Origin, 117–118
- OutlineOk, 258, 325–326
- Packages, 138
- PaintOval, 92
- PaintRect, 74, 90, 92
- ParamText, 294
- PasConsole, 42, 52
- PasInOut, 42, 44, 50, 316
- PasPrinter, 59
- PatCopy, 90
- Pattern, 73–74, 89
- Pause, 108, 164–165, 171
- Pen
 - location, 80
 - parameters, 78
 - pattern, 80, 82, 90, 92–93
 - size, 79, 116–117
 - transfer mode, 90
- PenMode, 78, 93–95
- PenPat, 48, 73–75, 80
- PenState, 77–78
- Picture handle, 216
- PICTURE.PAS, 212–216
- PICTURE.R, 212–213
- Pixel, 87, 110, 236, 238, 240, 257
 - in animation, 109
 - display methods, 92
 - examining, 119
 - font width, 210
 - in icon image, 349–350, 360
 - number between two points, 67, 176
 - number on display, 55
 - patterns, 74
 - pen size, 79
 - point size, 83
 - points on coordinate grid, 66
 - relation to bit map, 95
 - relation to memory bits, 56
- PnSize, 78
- Point, 67, 69–70, 77, 80
- Point size, 83–84, 86, 88, 178
- Pointer, 49, 51, 139. *See also* Master pointer
 - @ (at-sign), 75
 - assigning address, 51
 - converting, 52
- PortBits and PortRect, 101
- Printer. *See also* ImageWriter
 - driver, 361–362, 374
 - selecting features, 372
 - specification record, 370, 372, 375, 393
 - wide carriage, 372
- Printing. *See also* ImageUnit tools
 - changing default style, 392
 - characters per inch, 376, 393
 - characters per line, 370
 - embedded form feeds, 393
 - letter quality, 372
 - line numbers, 372
 - lines per page, 371
 - simulating form feeds, 372
 - specifying features, 369
 - text, 372
- PrnChar, 363, 367, 372
- PrnDispose, 363, 368, 373
- PrnEnd, 363, 369, 374
- PrnLine, 363, 368, 372
- PrnNew, 363, 368, 372
- PrnSetStyle, 363, 367, 372
- PrnStart, 363, 368, 373
- PrnString, 363, 367, 372
- Program Action, 123–125, 127, 140, 143, 171
- Program Engine, 126–128, 140, 146, 153–154, 203, 321–322
- PtInRect, 322
- PushButton, 325, 327, 330
- QuickDraw, 42, 44, 48, 55, 62, 64
 - line and pen tools, 78
 - oval tools, 90
 - round rectangle tools, 90
 - text, 82
- QUIT.PAS, 261–263
- QUIT.R, 260–261
- Radio button, 265, 328–330, 394
- RADIO.PAS, 267–270, 328, 330–331
- RADIO.R, 266–267
- RadioItem, 271
- Random, 49, 62–63, 99
- ReadChar, 27, 30, 46, 52
- READER.PAS, 217, 228–234
- READER.R, 217, 227–228
- Rect, 67, 70, 88, 90, 97
- Rectangle, 70, 89

- Refresh, 56
- Region, 69, 95, 97, 150, 211. *See also* Update region; Visible region
 - compared to rectangles, 73
 - disposing, 352
 - handle, 100
 - limiting with CopyBits, 108
 - starting new, 100
- REGIONS.PAS, 98–99
- ResEdit, 4, 6, 13, 18
- Reset, 28, 42, 50, 251, 285, 316
- Resize box, 162, 173
- ResizeWindow, 147, 165, 173
- Resource, 4, 160, 163, 177, 187
 - alert, 264–265
 - attribute, 160
 - binary file, 156
 - comments, 162
 - compiler, 3, 138
 - compiling text, 157
 - definition, 159, 162
 - dialog, 252, 256, 271, 277, 323
 - file, 136, 142, 157, 200, 290, 316, 343, 349, 392
 - icon, 350
 - icon list, 360
 - id, 257
 - menu, 138, 153
 - number, 122, 159
 - strings, 289
 - template, 178, 235
 - text, 128, 156, 159, 161, 294
- RETAB.PAS, 38–40
- Rewrite, 29, 50, 251, 285, 316
- RMaker, 3–4, 6, 13, 138, 156, 162–163, 203, 212, 252
- Rolling Rock, 70
- ROM version, 175, 210
- Round, 52
- RunProgram, 337–339, 343, 392

- Scan code, 147
- ScanEQ, 53
- ScanNE, 53
- ScreenBits, 79, 87, 100, 117, 173
- Scroll bar, 151–152, 162, 170, 175, 210, 217, 234, 240, 242–243
 - avoiding flutter, 152, 171, 239
 - dimming and highlighting, 235, 243
 - reforming, 243
 - resizing, 238
 - values, 244
- Scrolling, 236, 238, 241
 - automatic, 240, 243
- SelectIcon, 345, 348, 352
- SellText, 278
- Serial port, 210
- SetFPos, 46
- SetIText, 318–319
- SetOrigin, 117
- SetPenState, 77–78
- SetPort, 76
- SetRect, 69–70, 100
- SetVol, 43
- SF.PAS, 246, 248–250
- SF.R, 246–247
- SFGetFile, 339
- SFPutFile, 330
- SFReply, 250, 252, 338
- ShowCursor, 63
- ShowIconImage, 345–346, 350
- ShowIconMask, 345–346, 350
- ShowWindow, 162, 201
- SIN, 119
- SizeOf, 191, 318
- Stack, 182–183
- Stack/heap collision, 183, 186
- Standard file dialog, 138, 246, 250, 251, 252, 330, 338, 343, 375
 - limiting file types, 330
- STAR.PAS, 58, 63
- StartSound, 50
- StatText, 257, 271, 283, 316
- Stop mask, 179
- StopSound, 48
- STR#, 159
- String
 - replaceable areas in, 264
 - size on display, 84
- StringToNum, 50–51, 294, 319
- StringWidth, 84
- StuffHex, 106
- SwapWord, 53
- SysBeep, 141, 148
- System font, 178
- SystemEdit, 144
- SystemTask, 126, 155, 172, 294, 394

- Tab control character, 14, 371
- TABS.INC, 31–34, 36
- TEXT, 26–27, 34
- Text
 - font, 50, 82, 86
 - mode, 83
 - record, 242
 - streaming, 370–371
 - style, 84
- Text file, 26, 29, 50, 236, 244, 251
- Textbook program, 1, 8, 21–22, 43–45, 49, 136
 - display commands, 52
 - keypressed function, 99
 - standard library units, 72
- TextFont, 83
- TextHeight, 165, 168, 177
- TextMode, 93–94
- TEXTUNIT.PAS, 217–227

- Thumb box, 239–240, 243
- Time, 210, 374
- ToggleCheck, 324–325, 329
- TrackControl, 243
- Transfer mode, 90, 93, 95, 107
- TRANSFER.PAS, 336–338
- Transferring to programs, 20, 336
- Trunc, 52
- TUActivate, 218, 225, 243
- TUAttach, 218, 223, 235, 241
- TUClick, 218, 224, 235, 242
- TUDispose, 218, 224, 242
- TUHandle, 236
- TUReadText, 218, 226, 236, 244
- TURec, 236
- TUResize, 218, 226, 235, 243
- TurnRight, 62
- Turtle graphics, 44, 55, 57–58, 60, 62
 - plotting points, 48, 78
- Turtle unit, 58
- TUUpdate, 218, 225, 235, 243
- TWIRL.PAS, 60–61
- Type casting, 139, 142, 150, 176, 191
- Undo, 319
- Unit, 19, 179
 - compared to include file, 31
 - creating, 333
 - installing in compiler, 334–336
 - making temporary changes to, 335
 - number, 336
 - removing from compiler, 334
 - size, 336
- UnitMover, 4, 6, 137, 334–335
- UpCase function, 50–51
- Update event, 150–152, 212, 235, 243–244,
353, 393–394
- Update region, 152, 353
- UpString, 50
- USE\$, 59, 163, 235
- ValidRect, 152
- Variant record, 69
- VCenter, 325–326, 329
- Verified, 27
- Vertical axis, 65
- Vertical retrace interrupt, 141
- VHSelect, 67
- View rectangle, 241
- Visible region, 73, 152
- Volume number, 236, 244, 251, 392, 394
- Window. *See also* Alert; Dialog
 - closing, 176
 - creating, 177–178, 200–201
 - displaying contents, 243
 - dragging, 147
 - drawing contents, 150
 - erasing, 174
 - graphics in, 211
 - mouse click, 146
 - multiple, 194
 - pointer, 201
 - record, 236, 241–242, 257, 259
 - resizing, 147, 173–174
 - system, 146
 - zooming, 128, 175
- WindowPeek, 139–140, 176
- WindowPtr, 139, 145
- WindowRecord, 139
- Xor mode, 93–94
- Zoom box, 128, 147, 162, 173–174, 201, 235,
239
- Zooming shortcut, 16
- ZoomInOut, 165, 167, 174

Companion Disk Offer

To save time and avoid typing mistakes, you can order all of the programs in this book on disk for only \$30, postage paid. (For foreign orders, add \$5 for postage and handling.) Fill in and mail the order form below or, for faster service, telephone today. With your order, you'll receive:

- Two 3.5-inch disks suitable for any Macintosh computer
- Complete Pascal source code to all program listings
- All programs compiled and ready to run
- Additional instructions on disk

To order, call (717) 627-1911, 9 AM to 5 PM Eastern time, or write to:

Swan Software
Mac/Turbo Companion Disks
P.O. Box 206
Lititz, PA 17543

Name _____

Company _____

Address _____

City/St/Zip _____

Telephone _____

☐ Check or money order for \$30 enclosed (PA residents add 6% sales tax). Make checks payable to Swan Software.

☐ Bill my credit card ☐ Visa ☐ MasterCard

Card Number _____ Exp. Date _____

Signature _____

Price of \$30 includes postage to anywhere in the United States or to an APO number. Foreign orders add \$5 for postage and handling. Pennsylvania residents must add 6% sales tax (\$1.80). Allow 2 weeks for checks to clear.

...the ... of ...
...the ... of ...
...the ... of ...

...the ... of ...
...the ... of ...
...the ... of ...

...the ... of ...
...the ... of ...
...the ... of ...

...the ... of ...
...the ... of ...
...the ... of ...

...the ... of ...
...the ... of ...
...the ... of ...

...the ... of ...
...the ... of ...
...the ... of ...

...the ... of ...
...the ... of ...
...the ... of ...

...the ... of ...
...the ... of ...
...the ... of ...

...the ... of ...
...the ... of ...
...the ... of ...

...the ... of ...
...the ... of ...
...the ... of ...

Get more out of your Macintosh.

PROGRAMMING WITH MACINTOSH^{T.M.} TURBO PASCAL[®]

The Macintosh may be the easiest-to-use computer ever invented. But it's one of the toughest to program. And most Macintosh reference books aren't much help—you get pages and pages of what a program *does* before you find out how to do it yourself. What's the solution when you want to power up your Macintosh with added functions and features? *Programming With Macintosh Turbo Pascal*—a powerful toolkit of routines and utilities that teaches you good programming style as you put each new tool to work.

Author Tom Swan presents these tools in easy-to-use “library units” you can store on disk and continue to use long after you’ve mastered the basics of Pascal programming. Each unit is accompanied by descriptions, instructions, and numerous examples to help you master Turbo Pascal programming for the Macintosh without getting bogged down in details. After you’ve run through an application, you’ll not only know what tools are available and what they do, you’ll know exactly how they work. You get tools for building program shells and creating windows, tools for enhancing QuickDraw and Turtle-graphics, designing dialog boxes, reading and writing disk files, and powering up the Mac’s scroll bars, alerts, radio buttons and other controls.

Programming With Macintosh Turbo Pascal is an essential guide for every serious Macintosh user—for the professional programmer who needs quick, ready-to-run software tools on the job (it’s great for tight deadlines). For the business user who wants software tailored to his special requirements but who can’t afford to gamble time and money on a custom program. And for the student who wants to learn the tricks, techniques, and shortcuts the pros use.

TOM SWAN is the author of several computer books and has published dozens of articles in magazines such as *PC Tech Journal*, *inCider*, *Programmer’s Journal*, *PC World* and *Turbo Technix*.

JOHN WILEY & SONS

Business/Law/General Books Division
605 Third Avenue, New York, N.Y. 10158-0012
New York • Chichester • Brisbane • Toronto • Singapore

ISBN 0 471-62417-9